Inhaltsverzeichnis

Introduction **6**
Projekte 9
Challenges 15
Research 17
Fragen 18
Referenzen 18
Containers: Lists **20**
Projekte 24
Challenges 37
Research 41
Fragen 41
Referenzen 42
Containers: Maps & Sets **44**
Projekte 50
Challenges 54
Research 62
Fragen 62
Referenzen 63
Recursion **64**
Projekte 67
Challenges 78
Research 79
Fragen 80
Referenzen 81
Algorithmic Analysis **82**
Projekte 90
Challenges 93
Research 95
Fragen 96
Referenzen 97
Sorting **98**
Projekte 105
Research 107
Fragen 107
Referenzen 107
Trees **108**
Projekte 116
Challenges 135
Research 136
Fragen 137
Referenzen 138
Graphs **140**
Projekte 151
Challenges 159
Fragen 166
Referenzen 167
Text **168**
Projekte 174
Challenges 182
Research 183
Fragen 183
Referenzen 184
Techniques **186**
Greedy Algorithms 187
Backtracking 194
Randomized Algorithms 199
Review 201
Fragen 201
Referenzen 202

Variationen zum Thema: Algorithmen Eine Einführung anhand von Beispielen

von Ralph P. Lano, 1. Auflage

Für wen

das Buch richtet sich an Bachelor-Studierende im zweiten oder dritten Semester, aber auch an alle anderen die Spaß an kniffligen Problemen haben. Dieses Buch hat auch wieder viele Bilder, aber wie die Seitenanzahl andeutet, kommt auch hier viel Arbeit auf Sie zu. Aber wenn Sie das erste Buch [1] ausgehalten haben, dann schaffen Sie das hier auch!

Von wem

ich bin seit 2011 Professor für Internetprogrammierung und Multimediaapplikationen im Studiengang MediaEngineering an der Technischen Hochschule Nürnberg. Von 2003 bis 2010 war ich Professor für Softwaretechnik und multimediale Anwendungen an der Hochschule Hof, und von 2010 bis 2011 Professor für Media and Computing an der Hochschule für Technik und Wirtschaft Berlin. Ich promovierte 1996 an der University of Iowa zum Thema 'Quantum Gravity: Variations on a Theme'. Von 1996 bis 1997 war ich Postdoctoral Research Associate am Centre for Theoretical Studies des Indian Institute of Science. In der Zeit von 1997 bis 2003 war ich zunächst bei Pearson Education und später bei der Siemens AG in der Softwareentwicklung und dem Projektmanagement tätig.

Über was

Die Hauptthemen in diesem Buch sind Algorithmen und Datenstrukturen. Die Reihenfolge der Kapitel scheint vielleicht willkürlich, ist sie aber nicht. Ich gehe davon aus, dass die Leser im zweiten Semester sind, also gerade mal eine Vorlesung mit Java gehört haben und erste Schritte in der Objektorientierung unternommen haben, also ohne Stützräder programmieren können. Trotzdem fangen wir mit den einfachsten Algorithmen an, um das Thema etwas aufzuwärmen und uns etwas zu lockern. Dann könnte man zwar schon Rekursionen als nächstes machen, aber das würde viele abschrecken die nicht so mathematisch interessiert sind. Damit alle durchhalten und sehen, man kann da auch viel ohne Mathe machen, kommen die Datenstrukturen. Da gibt es viele schöne kleine Beispiele zu programmieren, die sind auch total nützlich und haben nichts mit Mathe zu tun. Hat man sich dann dran gewöhnt, dass es gar nicht so schlimm ist, kann man sich auch auf was Komplizierteres einlassen, die Rekursion. Die ist zwar erst ein bisschen verwirrend, aber es gibt da so viele coole Anwendungen, dass man dadurch motiviert wird sich mit der Materie auseinander zu setzen. Jetzt hat man dann hoffentlich soviel Blut geleckt, dass man sich auf die Algorithmische Analyse einlassen kann. Das ist wohl das trockenste Kapitel, aber auch das wichtigste. Danach geht's dann wieder bergab, weil obwohl die Konzepte leicht komplexer werden, sind die Beispielanwendung auch viel interessanter, und die Motivation kommt wieder von der Anwendungsseite. Dass Bäume vor Graphen kommen ist ganz klar. Und dass ich Bäume unbedingt benötige für die Konzepte wie Text, etc. ist auch klar. Zum Schluss beschäftigen wir uns noch mit ein paar sehr nützlichen algorithmischen Techniken. Und das war's dann auch schon, obwohl es eigentlich dann erst interessant würde...

Wie

lernt man mit Algorithmen umzugehen? Wie alles, durch viel üben! Deswegen ist auch dieses Buch wieder voll mit Übungsbeispielen. Die Veranstaltung so wie ich sie unterrichte besteht aus drei Komponenten: der Vorlesung, der Übung und Hausaufgaben. Die Vorlesung ist zwei Stunden pro Woche und entspricht jeweils dem ersten Teil eines Kapitels im Buch. Ein Kapitel schaffen wir in ca. ein bis zwei Wochen. In den Übungen, die zwei Stunden jede Wochen stattfinden, widmen wir uns dann den Projekten. Dabei schaffen wir zwischen zwei und vier der Projekte pro Übung. In der Übung arbeiten die Studierenden in Teams, meist zu zweit, um sich gegenseitig zu helfen. Die Hausaufgaben werden im zweiwöchentlichen Rhythmus bearbeitet und benötigen ca. 4 bis 5 Stunden. Es ist wichtig, dass die Studierenden alleine an der Hausaufgabe arbeiten.

Wo

finde ich die Beispiele und den Quellcode? Die gibt es auf der Webseite zum Buch: www.VariationenZumThema.de. Auch Updates, Links zur Entwicklungsumgebung, das Buch in elektronischer Version gibt's dort. Das Buch selbst gibt's bei Amazon, erst mal nur in Schwarz-Weiß (billig), die farbigen kauft eh keiner (teuer).

Darf ich

die Beispiele verwenden, oder das Buch kopieren? Dieses Material steht unter der Creative-Commons-Lizenz Namensnennung - Nicht-kommerziell - Weitergabe unter gleichen Bedingungen 4.0 International (CC-BY-NC-SA 4.0) D.h. Sie dürfen das Material in jedwedem Format oder Medium vervielfältigen und weiterverbreiten, das Material remixen, verändern und darauf aufbauen. Aber Sie müssen angemessene Urheber- und Rechteangaben machen, einen Link zur Lizenz beifügen und angeben, ob Änderungen vorgenommen wurden. Diese Angaben dürfen in jeder angemessenen Art und Weise gemacht werden, allerdings nicht so, dass der Eindruck entsteht, der Lizenzgeber unterstütze gerade Sie oder Ihre Nutzung besonders. **Sie dürfen das Material nicht für kommerzielle Zwecke nutzen.** Und wenn Sie das Material remixen, verändern oder anderweitig direkt darauf aufbauen, dürfen Sie Ihre Beiträge nur unter derselben Lizenz wie das Original verbreiten und Sie dürfen keine zusätzlichen Klauseln oder technische Verfahren einsetzen, die anderen rechtlich irgendetwas untersagen, was die Lizenz erlaubt. Um eine Kopie dieser Lizenz zu sehen, besuchen Sie http://creativecommons.org/licenses/by-nc-sa/4.0/. Der Quellcode steht unter der MIT License (http://choosealicense.com/licenses/mit/).

Warum

dieses Buch? Gibt es nicht schon genug gute Bücher zu Algorithmen? Auf jeden Fall, z.B. die Bücher von Sedgewick und Wayne [4], Roberts und Zelenski [5] und das Buch von Goodrich und Tamassia [6] sind total super. Natürlich auf Englisch. Und teuer. Aber auch sehr mathematisch angehaucht. Und das ist was dieses Buch versucht nicht zu sein: mathematisch. Vielmehr versucht es eher so ein Bilderbuch zu Algorithmen zu sein. Algorithmen sind echt cool, eigentlich gar nicht so schwer, meistens total intuitiv, aber man muss sie "sehen" und "begreifen" um sie zu verstehen, deswegen die vielen Bilder. Manchmal muss man sie auch gar nicht verstehen, sondern nur wissen wann, wo und wie man sie benutzt, oder welche gibt es denn überhaupt. Für die mathematischen Hintergründe, oder die genauen Herleitungen, gibt es ja schon tausende von guten Büchern. Warum wieder die ACM Library? Man könnte doch auch normales Java nehmen. Drei Gründe: wir kennen sie schon, sie ist total einfach, und erlaubt es uns uns auf das Wesentliche zu konzentrieren: die Algorithmen.

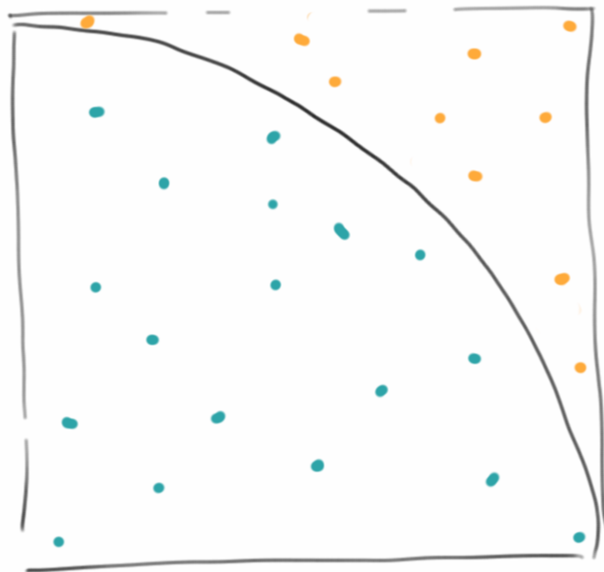
Woher

kommen die Ideen? Ohne Zweifel ist der Stil und der pädagogische Aufbau immer noch inspiriert von Mehran Sahami's Vorlesung [2], die ja wiederum auf dem Buch von Eric Roberts [3] basiert. Die Inhalte sind von mehreren Quellen inspiriert, vor allem der Vorlesung von Julie Zelenski [7] und dem Buch von ihr und Eric Roberts [5]. Aber auch das Buch von Goodrich und Tamassia [6] ist wirklich ausgezeichnet und voller exzellenter Beispiele. Vor allem ist es aber auch viel tiefer gehend als dieses hier. Wir kratzen hier nur ein bisschen an der Oberfläche. Noch ein Buch das sehr empfehlenswert ist, ist das von Sedgewick und Wayne [4]. Zu deren Buch gibt es auch einen Website und der ist wirklich voller wunderbarer Beispiele [8]. Ich kann all diese Bücher nur wärmstens empfehlen.

Referenzen

[1] Variationen zum Thema: Java: Eine spielerische Einführung, von Ralph P. Lano
[2] Programming Methodology, CS106A, von Mehran Sahami, https://see.stanford.edu/Course/CS106A
[3] The Art and Science of Java, von Eric Roberts, Addison-Wesley, 2008
[4] Introduction to Programming in Java, von Robert Sedgewick und Kevin Wayne
[5] Programming Abstractions in C++, von Eric S. Roberts und Julie Zelenski
[6] Data Structures and Algorithms in Java, von M.T. Goodrich und R. Tamassia
[7] Computer Science II: Programming Abstractions, von Julie Zelenski, Stanford, http://see.stanford.edu/see/courseinfo.aspx?coll=11f4f422-5670-4b4c-889c-008262e09e4e
[8] Introduction to Programming in Java, von Robert Sedgewick und Kevin Wayne, introcs.cs.princeton.edu/java/home/

Introduction



Algorithmen sind überall. Algorithmen bestimmen unser tägliches Leben. Beim Zusammenbauen von IKEA Möbeln, beim Backen eines Kuchens oder beim Spielen mit Lego folgen wir einem Algorithmus. Hinter selbstfahrenden Autos, Gesichtserkennung und Fingerabdrucksensoren stecken Algorithmen. Die Verschlüsselung, die künstlichen Intelligenz und auch die moderne Genetik verwenden Algorithmen. Es gibt Algorithmen die sind sehr alt, es gibt Algorithmen die sind sehr schön. Algorithmen können trivial sein, aber es gibt auch sehr komplexe Algorithmen. Es gibt sogar Algorithmen die produzieren Kunst. Auf den folgenden Seiten wollen wir uns ein wenig mit der Welt der Algorithmen vertraut machen, und wir beginnen ganz einfach.

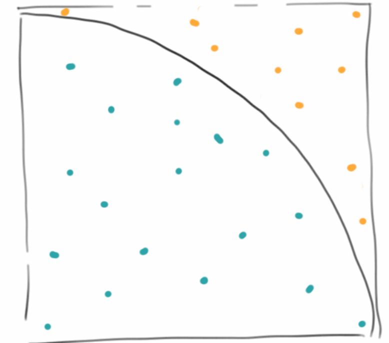
Introduction

Das Wort Algorithmus ist eigentlich die latinisierte Version des Nachnamens des persischen Mathematikers Abu Dscha'far Muhammad ibn Musa al-Chwarizmi (ca. 780 - ca. 850). Er ist u.a. verantwortlich für die Verbreitung des indischen Zahlensystems, einschließlich der Zahl Null. Er hat auch das Buch "Das kurzgefasste Buch über die Rechenverfahren durch Ergänzen und Ausgleichen" geschrieben, welches allgemein als Beginn der Algebra angesehen wird [3].

Laut Wikipedia ist ein Algorithmus eine effektive Methode etwas innerhalb einer begrenzten Zeit und mit begrenztem Raum zu berechnen, das man in einer wohl definierten Sprache ausdrücken kann. Diese Sprache beschreibt wie man von einem Anfangszustand, mit einer begrenzte Anzahl von Schritten, über wohl-definierte Zwischenzustände, schließlich einen finalen Endzustand erreicht [2].

Schauen wir uns einfach mal ein paar Beispiele an.

Pi



Es gibt viele Arten die Kreiszahl Pi auszurechnen, aber die einfachste ist wahrscheinlich die grafische: Wir zeichnen einfach ein Quadrat, und darin einen Viertelkreis. Dann malen wir einfach zufällig Punkte, möglichst gleichmäßig. Wir zählen die Punkte die innerhalb des Viertelkreises liegen, also *n_innerhalb = 18*, und die Gesamtzahl der Punkte, also *n_gesamt =28*. Pi ist dann einfach

```
Pi = 4 * n_innerhalb / n_gesamt = 2.6
```

Das ist jetzt nicht sehr genau, grob stimmt das aber. Wenn wir nämlich ganz viele Punkte machen, und die Punkte auch wirklich zufällig verteilt sind, dann kommt da wirklich Pi raus.

Longest Common Substring

DNA-Analyse hört sich jetzt an wie wenn es etwas super-kompliziertes wäre. Teilweise ist es das natürlich auch, speziell bis zu dem Schritt wo man die Buchstaben, also die Reihenfolge der Aminosäuren, hat. Hat man die aber, dann ist es ganz einfach. Nehmen wir an Bart's DNA sieht so aus:

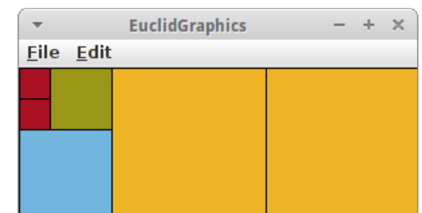
```
Bart = "GTTCCTAATA"
```

und die von Homer so

```
Homer = "CGATAATTGAGA".
```

Alles was man dann machen muss, um z.B. festzustellen ob Bart wirklich Homer's Sohn ist, die beiden DNA Sequenzen entlang der X- und Y-Achsen auf einem karierten Papier aufzutragen, und die Stellen an denen beide übereinstimmen, markiert man einfach mit einem Kreuzchen. Wenn man sich das dann ansieht findet man Kreuzchen die sich zu einer Diagonale verbinden. Je länger die Diagonalen, desto mehr Übereinstimmung zwischen den beiden DNAs [4].

Greatest Common Divisor (GCD)



Einer der ältesten Algorithmen der immer noch benutzt wird, stammt von Euclid [5]. Er beschreibt wie man den größten gemeinsamen Teiler zweier Zahlen findet. Die Anleitung wie man das macht, also der Algorithmus geht wie folgt:

1. ziehe die kleinere Zahl von der größeren solange ab, bis es nicht mehr geht;
2. ist die Zahl die übrig bleibt null sind wir fertig, die kleinere Zahl ist der größte gemeinsame Teiler;

3. falls die Zahl nicht null ist, dann machen wir aus der übrig gebliebenen Zahl die kleine Zahl und aus der kleinen Zahl die große und beginnen von vorne.

Am besten probiert man das einfach mal mit einem Beispiel aus: nehmen wir an die große Zahl ist 299, und die kleine Zahl ist 115. Verkürzt ergeben sich dann folgende Schritte:

```
299 = 115*2 + 69
115 = 69*1 + 46
 69 = 46*1 + 23
 46 = 23*2 + 0
```

Das bedeutet, dass 23 der größte gemeinsame Teiler von 299 und 115 ist.

Der Algorithmus hört sich ausgeschrieben etwas kompliziert an, aber wenn man ihn in Java übersetzt, dann ist er verblüffend einfach:

```
private int gcd(int a, int b) {
    while (b != 0) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

Der Algorithmus lässt sich übrigens auch sehr schön visualisieren.

Counting People

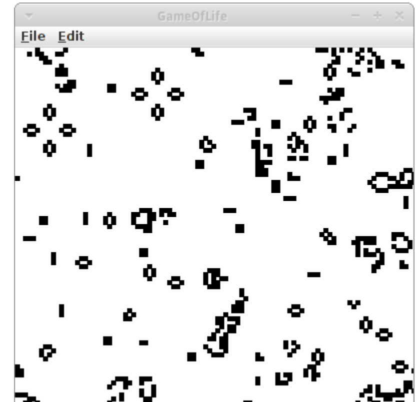
Wohl noch älter als Euclid's Algorithmus ist das Zählen. Man sollte meinen das ist ja eigentlich eine ganz einfach Angelegenheit. Aber sobald es sich um größere Mengen handelt wird es schon etwas schwieriger: wie zählt man denn die Anzahl der Zuschauer in einem etwas größeren Fußballstadium?

Wir könnten die Leute zählen wie sie ins Stadium kommen, einen nach dem anderen. Oder wir könnten einfach durchzählen lassen, wobei das in einem Stadium nicht ganz so einfach ist. Wir könnten auch ein Foto machen, und dann könnte einer die Leute auf dem Foto zählen. Auf die Art und Weise würden wir die genaue Anzahl der Leute wissen, solange wir uns nicht verzählen. Interessant ist die Frage wie lange das dauert: Nehmen wir an wir haben 20000 Leute im Stadium, und das Zählen einer Person dauert eine Sekunde. Dann dauert es knapp sechs Stunden bis wir fertig sind! Bis wir also fertig mit dem Zählen sind, ist das Spiel schon vorbei.

Stellt sich die Frage, geht das auch schneller? Wir könnten das Zählen "parallelisieren": wenn unser Stadium sechs Eingänge hätte, und wir würden an jedem Eingang zählen, dann würde es nur ein sechstel der Zeit in Anspruch nehmen. Wir wären also in einer Stunde fertig. Wenn wir 20000 Eingänge hätten, dann würde das Ganze nur eine Sekunde dauern!

Gibt es weitere Möglichkeiten Leute in einem Stadium zu zählen? Interessanterweise sehr viele. Vor allem wenn wir nur eine ungefähre Anzahl benötigen. Wir könnten z.B. durch den Stadiumsprecher bitten, dass die Leute deren Nachnahme mit 'A' losgeht aufstehen sollen. Die zählen wir dann und multiplizieren die Zahl mit 26. Schon 26 mal schneller. Wir könnten uns auch einen kleinen Teil des Stadiums suchen, von dem wir wissen wie viele Sitzplätze es dort gibt. Dann zählen wir wie viel Prozent der Sitzplätze dort besetzt sind. Wenn wir annehmen, dass im Schnitt die Leute gleichmäßig verteilt sind, können wir damit ausrechnen wie viele Leute ungefähr im Stadium sind. Wir könnten auch alle Sitze weiß anmalen, und den Leuten beim Eintritt rote Mützen geben. Dann machen wir ein Foto und messen einfach wie rot das Bild ist. Je roter desto mehr Leute. Oder wir fragen jemanden der sich mit Menschenmassen auskennt und wenn der dann sagt, "Oh, ich denke das sind so 20000", dann wäre das auch eine Möglichkeit zu zählen, aber eben nur eine Schätzung.

Worauf wir hinaus wollen: Sehr häufig gibt es mehr als nur einen Weg ein Problem zu lösen. Dabei sind manche Wege schneller als andere, andere sind dafür wieder genauer. Wenn wir also nach einem Algorithmus suchen, dann müssen wir zunächst entscheiden was uns wichtig ist, damit wir dann den passenden Algorithmus wählen können.



GameOfLife

Das größte Genie des letzten Jahrhunderts, John von Neumann, versuchte eine hypothetische Maschine zu konstruieren, die Kopien von sich selbst anfertigen konnte. Dies gelang ihm auch, allerdings hatte das mathematische Modell seiner Maschine sehr komplizierte Regeln. Dem britischen Mathematiker John Horton Conway schaffte es Anfang der 70er von Neumann's Ideen drastisch zu vereinfachen, heute bekannt unter dem Namen Conway's *Game of Life* [6].

Das Universum des Spiel des Lebens ist ein zweidimensionales Gitter aus quadratischen Zellen (GRects), von denen jede in einer von zwei möglichen Zuständen sein kann: lebend (schwarz) oder tot (weiß). Jede Zelle hat acht Nachbarn, und abhängig vom Zustand der Nachbarn entscheidet sich der eigene Zustand in der nächsten Runde nach folgenden Regeln:

- jede lebende Zelle mit weniger als zwei lebenden Nachbarn stirbt (Unter-Bevölkerung)
- jede lebende Zelle mit zwei oder drei lebenden Nachbarn lebt
- jede lebende Zelle mit mehr als drei lebenden Nachbarn stirbt (Über-Bevölkerung)
- jede tote Zelle mit genau drei lebenden Nachbarn wird eine lebende Zelle (Fortpflanzung)

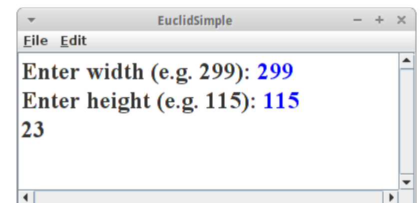
Das Resultat dieser einfachen Regeln ist durchaus überraschend.

Review

In der Einführung haben wir uns ein paar einfache, teilweise sehr alte Algorithmen angesehen. Wir haben auch gesehen, dass es Algorithmen gibt die nützlich sind und andere die nur eine Spielerei sind. Es gibt Algorithmen die schnell sind, und es gibt welche die langsam sind. Es gibt genaue und ungenaue Algorithmen. Und meistens gibt es mehr als einen Algorithmus ein bestimmtes Problem zu lösen. Das Wichtigste was allerdings hoffentlich rüberkommt: Algorithms are fun!

Projekte

Algorithmen sind so alt wie die Menschheit. Im Prinzip sind es einfach Kochrezepte, und genau wie diese sind die meisten Algorithmen auch intuitiv. Womit wir uns manchmal etwas schwer tun, ist sie dem Computer beizubringen. Aber wie mit allem, Übung macht den Meister.



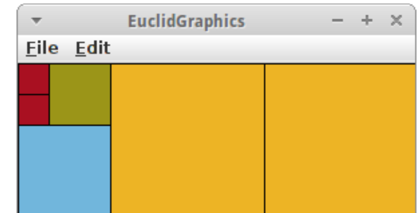
EuclidSimple

Wir beginnen mit der einfach Version, einem ConsoleProgram. Wir fragen den Nutzer nach zwei Zahlen,

```
int a = readInt("Enter width (e.g. 299): ");
int b = readInt("Enter height (e.g. 115): ");

println( gcd(a, b) );
```

und berechnen dann mit der Methode *gcd()* von oben den größten gemeinsamen Teiler.

EuclidGraphics



Als nächstes wollen wir den Euclidschen Algorithmus visualisieren. Dazu verwenden wir ein GraphicsProgram. Zunächst fragen wir wieder den Nutzer nach zwei Zahlen. Da es sich um ein GraphicsProgram handelt, verwenden wir die Klasse *IODialog*,

```
IODialog dialog = getDialog();
int w = dialog.readInt("Enter width (e.g. 299): ");
int h = dialog.readInt("Enter height (e.g. 115): ");

int x = gcd(w, h);
```

Mit dieser Klasse kann man auch Dinge ausgeben, z.B. mittels

```
dialog.println("GCD is:" + x);
```

Der Algorithmus selbst bleibt fast identisch

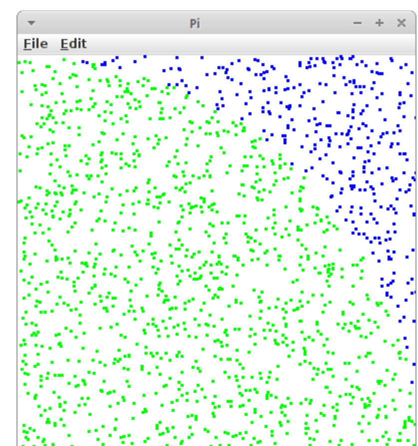
```
private int gcd(int a, int b) {
    while (b != 0) {
        if (a > b) {
            a = a - b;
            drawRect(a, 0, b, b);
        } else {
            b = b - a;
            drawRect(0, b, a, a);
        }
        pause(1000);
    }
    return a;

}
```

Wir müssen lediglich noch die Methode *drawRect()* implementieren. Die zeichnet einfach ein GRect mit zufälliger Farbe an den vorgegebenen Koordinaten:

```
private void drawRect(int a, int b, int w, int h) {
    GRect rect = new GRect(a, b, w, h);
    rect.setFilled(true);
    rect.setFillColor(rgen.nextColor());
    add(rect);

}
```

Gar nicht so schwer.

Pi



Wie bereits angedeutet kann Pi einfach durch das Zeichnen von Punkten ermitteln. Das mag zwar nicht die schnellste Methode sein, sie lässt sich aber am einfachsten visualisieren. Wir schreiben ein GraphicsProgram mit drei Instanzvariablen:

```
private RandomGenerator rgen =
RandomGenerator.getInstance();
private int totalPoints = 0;
private int insidePoints = 0;
```

einen Zufallszahlengenerator, und zwei Zählern, einen für die Gesamtzahl der Punkte und einen für die Punkte die innerhalb des Viertelkreises liegen. In der *run()* Methode zeichnen wir dann jeweils einen Punkt, und berechnen nach jedem Mal Pi und geben es auf der Konsole aus:

```
while (true) {
    drawRandomPoint();
    double pi = 4.0 * insidePoints / totalPoints;
    System.out.println( "Pi = " + pi );

}
```

Die *drawRandomPoint()* Methode macht auch nicht besonders viel:

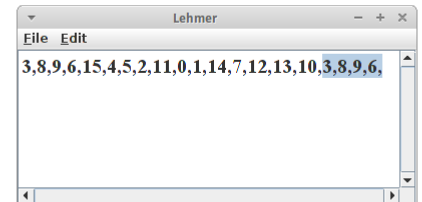
```
private void drawRandomPoint() {
    double x = rgen.nextDouble();
    double y = rgen.nextDouble();
    totalPoints++;
    GRect point = new GRect(x*SIZE,SIZE-y*SIZE, 1,1);
    if ( ( x*x+y*y ) < 1.0 ) {
        insidePoints++;
        point.setColor( Color.RED );
    } else {
        point.setColor( Color.BLUE );
    }
    add( point );

}
```

Wir holen uns zwei zufällige Werte, x und y, erhöhen unseren Punktezähler und generieren ein Punkt Objekt. Da es in der ACM Library keine Punkte gibt, nehmen wir einfach ein GRect, das eine Höhe und Breite von je eins hat. Geht auch. Der Trick ist, wie weiss ich ob ein Punkt nun innerhalb oder außerhalb des Viertelkreises ist? Falls wir das vergessen haben, schlagen wir das schnell in einem alten Mathebuch nach: Alle Punkte bei denen

```
( x*x + y*y ) < 1.0
```

die sind innerhalb des Kreises. Und die malen wir rot an, die anderen blau. Funktioniert ganz gut. Falls man mehr über Pi wissen möchte, kann man sich das Buch eines Kollegen kaufen [7].

Lehmer



Wie wir gerade gesehen haben können Zufallszahlen ganz praktisch sein. Auch letztes Semester haben wir schon häufiger einen Zufallszahlengenerator, den *RandomGenerator*, verwendet. Nur wie funktioniert der, wo kommen denn die Zufallszahlen her?

Wenn man Glück hat, kommen die aus der Natur. Ein Computer hat jetzt aber recht wenig mit Natur zu tun, und für einen Computer ist es überraschend schwer an gute Zufallszahlen heranzukommen. Aber Gott sei Dank gab es den Herrn Lehmer und der hat sich da was überlegt, den Lehmer Algorithmus [8]:

```
X_i+1 = ( a * X_i + c ) % m
```

dabei ist '%' unser Freund der Modulo Operator. Der Lehmer Algorithmus erzeugt Pseudo-Zufallszahlen zwischen 0 und m-1, die linear kongruent sind, also gleichmäßig verteilt. Die Konstanten a, c und m müssen zwei Bedingungen erfüllen

```
2 <= a < m     und     0 <= c < m
```

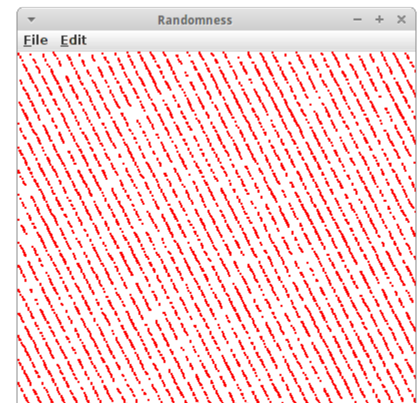
z.B., a=13, c=1, und m=16. Das erste x, also x_0, kann beliebig sein, idealerweise aber auch kleiner als m. Man nennt dieses erste x auch das *Seed*.

Wenn wir den Algorithmus in Java übersetzen

```
int a = 13;
int c = 1;
int m = 16;
int x = 3;

for (int i = 0; i < 20; i++) {
    print(x + ",");
    x = (a * x + c) % m;
}
```

und mal ausprobieren, dann stellen wir etwas interessantes fest: die Zahlen wiederholen sich! Und zwar nach m Schritten. Das ist der Grund warum diese Zahlen auch Pseudo-Zufallszahlen nennt. Es stellt sich heraus, dass alle im Computer durch Algorithmen erzeugten Zufallszahlen immer Pseudo-Zufallszahlen sind.

Kann man da was machen? Nein. Aber durch geschickte Wahl der Konstanten a, c und m kann man das Ganze erträglich machen.



Randomness

Was ist denn ein geschickte Wahl für der Konstanten a, c und m? Man kann jetzt Mathematik studieren (durchaus empfohlen) oder man liest in schlauen Büchern nach [9]. Wenn man relativ gute Pseudo-Zufallszahlen auf einem 32 bit Computer erzeugen will, dann sind folgende gute Werte:

- wenn man für m eine Primzahl wählt, dann kann man $c = 0$ setzen;
- die Primzahl m sollte möglichst groß sein, und wir haben Glück: $2^{31} - 1$ ist eine Primzahl;
- und die Mathematiker erzählen uns, dass $a = 7*7*7*7*7 = 16807$ eine gute Wahl für a ist.

Diese Wahl nennt man auch den "Minimal Standard Random Number Generator".

Schauen wir mal ob er was taugt, also ob er gut genug ist. Es stellt sich nämlich heraus, dass unser Auge relativ gut erkennen kann ob etwas zufällig ist oder nicht. Es braucht nur ein klein bisschen Unterstützung. Als erstes implementiern wir unseren Lehmer Algorithmus, oder genauer den "Minimal Standard Random Number Generator" in Java:

```
private long a = 7 * 7;  // * 7 * 7 * 7; // use 7 or 7*7
private long m = 2147483647L;
private long x = System.currentTimeMillis();

public int nextInt() {
    x = a * x % m;
    return (int) x;
}
```

Für unsere Seed x verwenden wir einfach die Uhrzeit, damit bekommen wir jedes mal andere Zufallszahlen. Macht man echt so. Ist auch der Grund warum die NSA so einfach unsere Verschlüsselungen knacken kann. Die Methode *nextInt()* erzeugt Zufallszahlen zwischen 1 und 2^31 (-2). Wir benötigen aber sehr häufig Zahlen zwischen 0 und einer Obergrenze n:

```
public int nextInt(int n) {
    int z = nextInt();
    return (int) (z % n);
}
```

und da ist er wieder unser Freund der Modulo Operator.

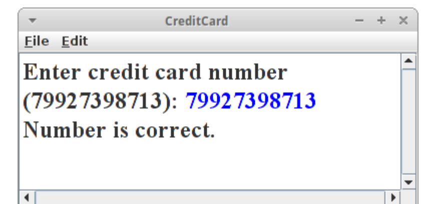
Kommen wir zum Auge. Dafür schreiben wir ein GraphicsProgram in dessen *run()* Methode wir ein paar zufällige Punkte malen:

```
for (int i = 0; i < 10000; i++) {
    int x = nextInt(SIZE);
    int y = nextInt(SIZE);
    setPixel(x, y, Color.RED);

}
```

wobei *setPixel()* wieder wie oben beim Project Pi ein GRect mit einem Pixel Höhe und Breite malt.

Wenn wir uns das Resultat ansehen, und *a* auf den Wert 7*7 gesetzten haben, dann sehen wir rote Streifen! Unser Auge sagt uns, das sind keine guten Zufallszahlen. Wählen wir aber für *a* den Wert den uns die Mathematiker nahelegen, also 7*7*7*7*7, dann können wir keine Streifen mehr erkennen. Keine Streifen, oder irgendwelche Muster im Allgemeinen, sind ein Zeichen für einen guten Zufallszahlengenerator.

CreditCard



Wie weiss man ob man sich vertippt hat? Dafür gibt es Prüfsummen (checksum). Ein Beispiel ist der Luhn-Algorithmus, von dem deutsch-amerikanischen Informatiker Hans Peter Luhn [10], der z.B. bei Kreditkarten verwendet wird.

Eine Kreditkarte besteht aus 16 Zahlen. Dabei sind die ersten 15 die eigentliche Nummer, die letzte Zahl ist aber die sogenannte Prüfziffer. Wie funktioniert der Luhn-Algorithmus?

- Erst mal wird jede zweite Ziffer verdoppelt, beginnend bei der zweiten von rechts. wenn dieses Resultat größer als neun ist, wird neun abgezogen;
- dann werden alle Ziffern aufaddiert, also sowohl die nicht verdoppelten als auch die verdoppelten, wenn diese Summe modulo 10 die Null ergibt, ist alles in Ordnung.

Eine Implementierung in Java sieht wie folgt aus:

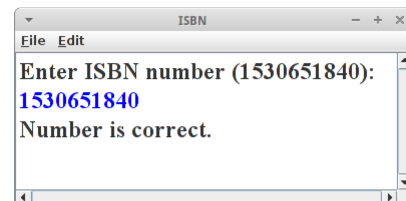
```
private boolean checkCreditCardNumber(String creditNumber) {
    int sum = 0;
    int len = creditNumber.length();
    for (int i = 0; i < len; i++) {
        int x = creditNumber.charAt(i) - '0'; // turn char in to int
        int y = x * (2 - (i + len) % 2);      // multiply by two every
other
        if (y > 9) {
            y -= 9;
        }
        sum += y;
    }
    return sum % 10 == 0;

}
```

Es ist interessant zu sehen wie die Anforderung "die zweite Ziffer von rechts verdoppeln" umgesetzt wurde:

```
int y = x * (2 - (i + len) % 2);
```

Das muss man einfach mal auf Papier ausprobiern, und dann sieht man, dass das anscheinend funktioniert.

ISBN



Ähnlich wie bei Kreditkarten gibt es auch bei Büchern die Internationale Standardbuchnummer (ISBN) anhand der man Bücher eindeutig identifizieren kann [11]. Auch bei dieser Nummer gibt ist die letzte Ziffer eine Prüfziffer. Der Algorithmus ist sogar noch einfacher als bei den Kreditkarten:

- addiere jede Ziffer multipliziert mit ihrer Position;
- wenn das Resultat modulo 11 die Null ergibt stimmt die Nummer.

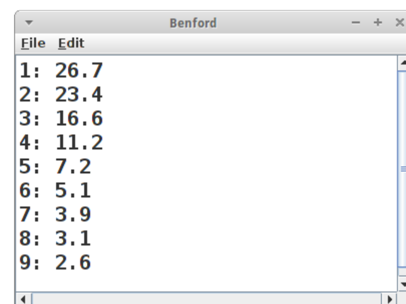
In Java wird daraus:

```
char[] arr = isbnNumber.toCharArray();
for (int i = 0; i < 9; i++) {
    t = arr[i] - '0';
    s += t * (i + 1);

}
```

Eine kleine Ausnahme gibt es: die Prüfziffer könnte eine 10 sein, dann macht man einfach ein 'X' daraus, also 111111112X:

```
if (arr[9] == 'X') {
    s += 10 * 10;
} else {
    t = arr[9] - '0';
    s += t * 10;
}

if (s % 11 == 0) { ... }
```

Benford's Law



Gerade haben wir gesehen wie man feststellen kann ob in Kreditkarten- oder ISBN Nummern ein Fehler ist. Dafür verwendet man einen Algorithmus. Kann man aber auch in anderen Daten Fehler finden? Ein interessantes Beispiel dafür ist das Benfordsche Gesetz [12]. Eigentlich hat es zuerst ein Herr Newcomb entdeckt, und deswegen nennt man es auch manchmal *Newcomb-Benford's Law*.

Es geht darum, dass es in empirischen Daten eine überraschende Regelmäßigkeit gibt mit der gewisse Ziffern auftreten, speziell die erste Ziffer. Naiv würde man erwarten, dass alle Ziffern gleich oft dran kommen, und für zufällig verteilte Daten ist das auch der Fall. Aber eben nicht für die meisten empirischen Daten. Die folgen nämlich sehr häufig der folgenden Verteilung:

```
'1'    30,1 %
'2'    17,6 %
'3'    12,5 %
'4'     9,7 %
'5'     7,9 %
'6'     6,7 %
'7'     5,8 %
'8'     5,1 %
'9'     4,6 %
```

(Quelle [12]). D.h. 30 Prozent aller Zahlen in einem empirischen Datensatz beginnen mit der Ziffer '1'. Steuerfahnder verwenden diese Gesetzmäßigkeit um Steuerbetrug aufzudecken, und auch Wahlbetrug ist schon auf diese Art und Weise entlarvt worden.

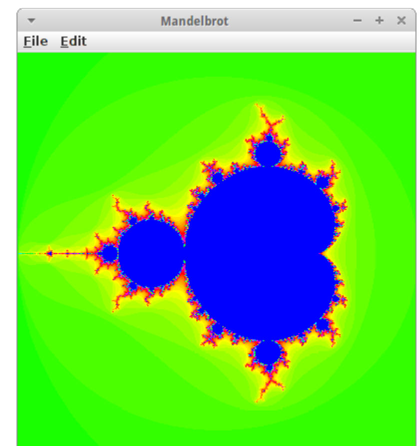
Im letzten Semester hatte wir schon einmal mit empirischen Daten zu tun und das waren Aktienkurse. Im Prinzip können wir die Klasse *StockDataBase* unverändert verwenden. Dann sind in der *stockDB* HashMap die ganzen Aktienkurse gespeichert. Die gehen wir dann einen nach dem anderen durch und zählen wie häufig sie mit einer bestimmten Ziffer beginnen. Das Zählen machen wir in dem Array *counts*.

```
public double[] analyze() {
    double total = 0;
    double[] counts = new double[10];
    for (String stock : stockDB.keySet()) {
        StockEntry ent = stockDB.get(stock);
        List<Double> prices = ent.getPrices();
        for (int i = 0; i < prices.size(); i++) {
            double price = prices.get(i);
            char c = String.valueOf(price).charAt(0);
            if (Character.isDigit(c)) {
                counts[c - '0']++;
                total++;
            }
        }
    }
    return counts;
}
```

Danach sollten wir noch die Daten in Prozent umrechnen, damit wir sie mit dem Benfordsche Gesetz vergleichen können. Und interessanterweise scheinen auch Aktienkurse grob dem Benfordsche Gesetz zu folgen. Ob der Unterschied vielleicht auf Insider-Trading hindeutet?

Das Benfordsche Gesetz ist natürlich kein Algorithmus. Deswegen stellt sich natürlich die Frage gehört das in dieses Buch. Die Frage darf jeder für sich selbst beantworten. Auf den Webseiten zum Buch von Sedgewick und Wayne finden sich noch eine ganze Menge anderer Datensätze über die man ähnliche Analysen laufen lassen könnte [13].

Challenges



Mandelbrot

Die Apfelmännchen sind nach dem französischen Mathematiker Benoît Mandelbrot benannt. Es handelt sich dabei um sogenannte Fraktale, aber die meisten Leute finden sie einfach nur hübsch [14].

Die mathematische Gleichung die hinter der Mandelbrot Menge liegt ist sehr einfach:

```
z_n+1 = z_n * z_n + c
```

dabei sind z und c komplexe Zahlen. Es handelt sich hier um eine Iteration, d.h. wenn wir z_n kennen, dann können wir z_{n+1} ausrechnen. Die Anfangsbedingungen lauten, dass z_0 gleich null sein soll und c ist der Punkt in der komplexen Ebene für den die Farbe ausgerechnet werden soll. Also wenn wir in x- und y-Koordinaten denken, dann ist

```
c = x + i y
```

die Anfangsbedingung. Alles was noch nötig ist, ist das Abbruchkriterium, wann sollen wir mit der Iteration aufhören? Entweder wenn z*z >= 4 ist oder wenn die Anzahl der Iterationen größer als ein maximal Wert ist:

```
while ( (x*x + y*y < 4) && (iteration < max_iteration) ) {
    ...
    iteration++;

}
```

Damit das Ganze dann hübsch aussieht, nehmen wir die Anzahl der Iterationen und kodieren sie in Farbe:

```
int color = RAINBOW_COLORS[iteration % RAINBOW_NR_OF_COLORS];
```

Dabei ist *RAINBOW_COLORS* ein Farbarray, das wir beliebig initialisieren können. Zu guter Letzt brauchen wir noch eine *setPixel()* Methode, die es in der ACM Graphics Bibliothek eigentlich gar nicht gibt. Wir behelfen uns damit, dass wir kleine GRects zeichnen:

```
private void setPixel(double x, double y, Color color) {
    int i = (int) (((x - xMin) * WIDTH) / (xMax - xMin));
    int j = (int) (((y - yMin) * HEIGHT) / (yMax - yMin));
    GRect r = new GRect(1, 1);
    r.setColor(color);
    add(r, i, j);

}
```

Das ist nicht gerade die schnellst und effektivste Art, aber sie funktioniert.

RandomGenerator

Letztes Semester haben wir häufig die Klasse RandomGenerator der ACM Bibliothek benutzt. Inzwischen können wir diese Klasse selbst implementieren. Dazu benutzen wir einfach Lehmer's Algorithmus mit den Konstanten wie sie z.B. in Referenz [8] empfohlen werden:

```
public final class RandomGenerator {
    private int seed = 1;

    private static final int a = 16807; // = 7*7*7*7*7
    private static final int m = 2147483647; // = 2^31 -1
    private static final int q = 127773; // = m div a
    private static final int r = 2836; // = m mod a

    public RandomGenerator() {
    }

    /**
     * @return a random number between 0 and 2147483647
     */
    public int nextInt() {
        seed = a * (seed % q) - r * (seed / q);
        return seed;
    }

    /**
     * sets the initial seed
     *
     * @param s
     *            a good idea is a changing value, such as the time,
     *            ideally it is a truely random number.
     */
```

```
    public void setSeed(int s) {
        if ((s < 1) || (s >= m)) {
            throw new IllegalArgumentException("invalid seed");
        }
        seed = s;
    }

}
```

Was jetzt noch zu tun bleibt sind die übrigen Methoden der RandomGenerator Klasse zu implementieren, z.B.:

- **int nextInt(int n):** gibt eine zufällige Ganzzahl zwischen 0 <= r < n zurück;
- **int nextInt(int low, int high):** gibt eine zufällige Ganzzahl zwischen low <= r < high zurück;
- **boolean nextBoolean():** gibt einen zufällige Boolean mit einer 50/50 Wahrscheinlichkeit von true oder false zurück;
- **double nextDouble():** gibt eine zufällige Gleitkommazahl zwischen 0 <= r < 1 zurück;
- **double nextDouble(double low, double high):** gibt eine zufällige Gleitkommazahl zwischen low <= r < high zurück;
- **Color nextColor():** gibt eine zufällige Farbe zurück.

Research

Natürlich können wir in diesem Buch nur die Oberfläche streifen, aber es gibt einige Themen die man noch vertiefen könnte.

List of Algorithms

Um eine Vorstellung davon zu bekommen, wie viele Algorithmen es da draußen gibt, werfen wir einen Blick auf die Sammlung von Algorithmen, die in der Wikipedia gelistet werden [1].

Entscheidungsproblem

David Hilbert's *Entscheidungsproblem* und Alan Turing's *Turing Machine* sind zwei Themen über die wir mal recherchieren sollten.

Luhn vs ISBN

Wenn wir den Luhn Algorithmus mit dem ISBN Algorithmus vergleichen, stellt sich die Frage: Welcher ist besser? Welcher erkennt mehr Fehler, bzw. häufiger auftretende Fehler?

Fragen

1. Wer hat den ersten Algorithmus geschrieben?
2. Beschreiben Sie in Ihren eigenen Worten, wie der Größte Gemeinsame Teiler Algorithmus funktioniert.
3. Nennen Sie zwei der wichtigsten Errungenschaften von Muhammad ibn Musa al-Chwarizmi.
4. Was ist der Unterschied zwischen einer echten Zufallszahl und einer Pseudozufallszahl?
5. Wie kann man die Kreiszahl Pi mit zufälligen Zahlen berechnen?
6. Beschreiben Sie die grafische Version von Euclid's Algorithmus.
7. Wofür wird Lehmer's Algorithmus verwendet?
8. Wie können Sie grafisch bestimmen, ob ein Zufallszahlengenerator schlecht ist?

Referenzen

Anbei finden sich die Referenzen zum ersten Kapitel.

[1] List of algorithms, en.wikipedia.org/wiki/List_of_algorithms

[2] Algorithm, en.wikipedia.org/wiki/Algorithm

[3] Muḥammad ibn Mūsā al-Khwārizmī, https://en.wikipedia.org/wiki/Muhammad_ibn_Musa_al-Khwarizmi

[4] Longest common substring problem, https://en.wikipedia.org/wiki/Longest_common_substring_problem

[5] Euclidean algorithm, https://en.wikipedia.org/wiki/Euclidean_algorithm

[6] Conways Spiel des Lebens, https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

[7] Pi: Algorithmen, Computer, Arithmetik, Jörg Arndt, Christoph Haenel

[8] Lehmer random number generator, https://en.wikipedia.org/wiki/Lehmer_random_number_generator

[9] Generating Random Numbers in Data Structures and Algorithms, Bruno R. Preiss, http://www.brpreiss.com/books/opus5/html/page465.html#33557

[10] Luhn-Algorithmus, https://de.wikipedia.org/wiki/Luhn-Algorithmus

[11] International Standard Book Number, https://en.wikipedia.org/wiki/International_Standard_Book_Number#ISBN-10_check_digits

[12] Benfordsches Gesetz, https://de.wikipedia.org/wiki/Benfordsches_Gesetz

[13] Real-World Data Sets, Robert Sedgewick and Kevin Wayne, http://introcs.cs.princeton.edu/java/data/

[14] Mandelbrot-Menge, https://de.wikipedia.org/wiki/Mandelbrot-Menge

Containers: Lists

Nach unserem ersten Kontakt mit ein paar Algorithmen, wollen wir uns jetzt mit Containern, also Behältern beschäftigen. Mit Behältern meinen wir Behälter für Daten, also Datencontainer. Grundsätzlich gibt es zwei Arten:

- sequentielle Container und
- assoziative Container.

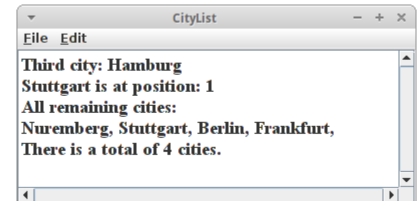
Beispiele für sequentielle Container sind Arrays, ArrayList und LinkedList, sowie die Stack und die Queue Klasse. Beispiele für assoziative Container sind die Maps und die Sets. In diesem Kapitel beschäftigen wir uns mit den sequentiellen Container.

Arrays

Den einfachsten Container den wir bereits aus dem ersten Semester kennen ist das Array. Wenn wir z.B. ein Array für zehn Ganzzahlen anlegen wollen, dann geht das so:

```
int[] eggs = new int[10];
```

Arrays sind zwar sehr schnell, haben aber den Nachteil, dass sie ihre Größe nicht mehr ändern können. Wenn wir also einmal ein Array für zehn Elemente angelegt haben, dann können wir da keine elf rein tun. Wir verwenden Arrays eigentlich nur wenn wir es mit Binärdaten zu tun haben, z.B. Bilder-, Audio- oder Videodateien. Ansonsten werden wir Arrays eher meiden, denn es gibt viel bessere Datenstrukturen.



Lists

Die Liste ist wahrscheinlich der nächst einfachste Datencontainer nach dem Array. Die Liste existiert in mehreren Varianten, die beiden prominentesten sind die *ArrayList* und die *LinkedList*. Eine Liste ist ein sequentieller Container, d.h. die Elemente einer Liste haben eine Ordnung, immer beginnend bei Index 0, und es gibt keine Löcher in einer Liste. Eine Liste hat die folgenden Methoden:

- **size():** gibt die Größe der Liste zurück;
- **add(object):** fügt ein Objekt am Ende der Liste an;
- **get(i):** gibt das Objekt, das an Position i gespeichert ist;
- **set(i, object):** ersetzt das Element an Position i mit diesem neuen Objekt;
- **remove(i):** das Element an Position i wird entfernt;
- **indexOf(object):** sucht nach dem Objekt in der Liste und gibt dessen Position zurück, oder -1 falls es nicht in der Liste ist. Diese Methode ist nicht die schnellste wie wir noch sehen werden.

Es gibt auch noch eine *contains()* Methode, die sollten wir aber eher selten benutzen.

Schauen wir uns mal ein Beispiel an wie man Listen im echten Leben verwendet. Angenommen wir hätten mehrere Städte die wir nacheinander bereisen wollten, dann wäre eine Liste eine gute Wahl:

```
// init list
List<String> cities = new ArrayList<String>();
// add cities
cities.add("Nuremberg");
cities.add("Munich");
cities.add("Hamburg");
cities.add("Berlin");
cities.add("Frankfurt");

// get the third city
println("Third city: " + cities.get(2));

// remove the third city
cities.remove(2);

// replace the second city by another city
cities.set(1, "Stuttgart");

// search for Stuttgart
println("Stuttgart is at position: " + cities.indexOf("Stuttgart"));

// list all remaining cities
println("All remaining cities:");
for (String city : cities) {
    print(city + ", ");
}
println("\nThere is a total of " + cities.size() + " cities.");
```

Wann sollten wir Listen verwenden? Listen können verwendet werden, wenn

- die Reihenfolge wichtig ist, z.B. wenn man von Stadt zu Stadt reist: München -> Nürnberg -> Berlin -> Hamburg;
- man eine Datenstruktur benötigt, die einen Index und einen zugehörigen Wert hat;
- man ein Array benötigt, das dynamisch schrumpfen und wachsen kann.

List Interface

Vielleicht ist aufgefallen, dass wir oben nicht

```
ArrayList<String> cities = new ArrayList<String>();
```

verwendet haben, sondern anstelle

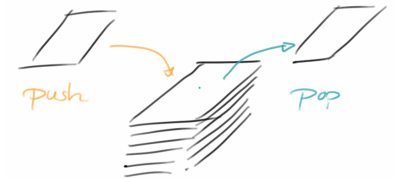
```
List<String> cities = new ArrayList<String>();
```

Das war absichtlich. Warum? Der Grund dafür ist, dass sowohl *ArrayList* als auch *LinkedList* das List *Interface* implementieren. Das ist sehr praktisch, denn wenn wir aus irgendeinem Grund später unsere Meinung ändern und eine *LinkedList* anstelle einer *ArrayList* verwenden wollen, gibt es nur eine Stelle die wir ändern müssen. Wir werden diese Verwendung von Schnittstellen später immer wieder sehen.

Stellt sich die Frage, wann soll ich eine *ArrayList* verwenden und wann eine *LinkedList*? Die Antwort ist ziemlich einfach:

- wenn wir viel lesen, verwenden wir die ArrayList,
- wenn wir aber viel schreiben (also hinzufügen, ersetzen oder entfernen), dann verwenden wir besser die LinkedList.

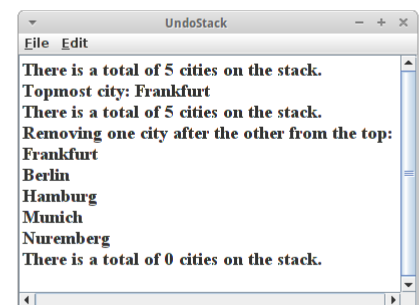
Stack (LIFO)



Kommen wir zu einer neuen Datenstruktur, dem *Stack*. Wenn man sich unter Stack einen Stapel Papier vorstellt, dann hat man die Datenstruktur eigentlich schon verstanden. Bei einem riesen Stapel Papier kann man nicht einfach ein Blatt aus der Mitte herausziehen, das geht praktisch nicht. Genauso ist es auch beim Stack: wir können nur auf das oberste Element zugreifen. Erst wenn wir das oberste Element wegnehmen, können wir sehen was darunter ist. Oder wenn wir neue Elemente hinzufügen, können wir die nur oben hinzufügen. Wir können nicht einfach zwischendrin irgend ein Blatt einfügen. Das führt zu der Bezeichnung Last-In-First-Out (LIFO).

Welche Methoden braucht man um mit dem Stack arbeiten zu können? Im Prinzip genügen die folgenden:

- **size():** gibt die Größe des Stacks zurück, also wieviel Objekte im Stack sind;
- **push(object):** legt ein neues Objekt oben auf dem Stapel ab;
- **pop():** nimmt das oberste Element vom Stapel, das Element ist danach nicht mehr im Stapel;
- **peek():** gibt uns das oberste Element des Stapels, lässt es aber im Stapel.



Als kleine Anwendung betrachten wir folgendes Beispiel: Nach unserer Reise quer durch Deutschland, wollen wir in genau umgekehrter Reihenfolge wieder zurückreisen. Genau dafür eignet sich der Stapel perfekt:

```
// init stack
Stack<String> cities = new Stack<String>();

// add cities to the stack
cities.push("Nuremberg");
cities.push("Munich");
cities.push("Hamburg");
cities.push("Berlin");
```

```
cities.push("Frankfurt");
println("There is a total of " + cities.size() + " cities on the
stack.");

// whats on top of the stack?
println("Topmost city: " + cities.peek());
println("There is a total of " + cities.size() + " cities on the
stack.");

// remove one by one the top element from the stack
println("Traveling back the way we came:");
println(cities.pop());
println(cities.pop());
println(cities.pop());
println(cities.pop());
println(cities.pop());

println("There is a total of " + cities.size() + " cities on the
stack.");
```

Wofür sonst kann man denn die Datenstruktur Stack noch verwenden? Z.B. für:

- Browser History: man möchte die zuletzt besuchten Webseiten in umgekehrter Reihenfolge auflisten;
- man möchte die Reihenfolge von irgendetwas umkehren;
- in einem Editor (z.B. Word oder Eclipse) wird ein Stack verwendet für die Undo Aktionen;
- in der Mathematik oder beim Programmierung möchte man sicher stellen, dass es für jede offene Klammer wieder eine geschlossene gibt;
- in einer sogenannten Stack-Machine (z.B. die Java Virtual Machine ist eine Stack-Machine).

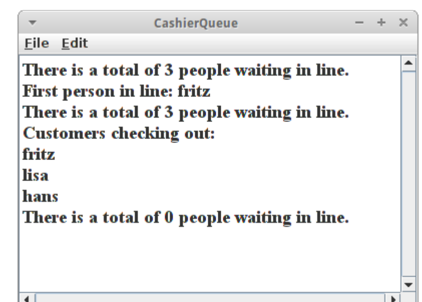
Im Deutschen findet man auch manchmal für die Datenstruktur Stack die Bezeichnung *Keller*, macht wenig Sinn, klar Dachgeschoß würde viel mehr Sinn machen.

Queue (FIFO)

Die *Queue*, auch Warteschlange genannt, ist eine Liste die dem Prinzip des First-In-First-Out folgt. Ähnlich dem Prinzip einer Warteschlange an der Kasse, oder wenn man in einer Telefonwarteschlange ist, erwartet man, dass die Personen die weiter vorne in der Schlange stehen zu erst dran kommen. Für unsere Datenstruktur bedeutet das, dass man nur auf der linken Seite (dem Ende der Schlange) neue Elemente hinzufügen kann, und nur auf der rechten Seite (dem Anfang der Schlange) Elemente entnehmen kann.

Die folgenden Methoden werden von der *Queue* Datenstruktur unterstützt:

- **size():** gibt uns die Anzahl der Elemente in der Queue;
- **add(object):** fügt ein neues Element am Ende der Schlange hinzu;
- **remove():** entnimmt ein Element am Anfang der Schlange;
- **peek():** ähnlich wie beim Stack erlaubt uns *peek()* nachzusehen was denn am Anfang der Schlange ist, ohne das Element aber von der Schlange zu entfernen.



Ein bisschen komisch ist, dass es in Java keine Klasse Queue gibt, sondern nur ein Interface. Ist aber auch nicht so schlimm. Zwei Klassen die dieses Interface implementieren sind die *LinkedList* und die *PriorityQueue*.

Als kleines Beispiel setzen wir eine Warteschlange vor einer Kasse um:

```
// init queue
Queue<String> lineOfPeople = new
LinkedList<String>();
```

```
// add people
lineOfPeople.add("fritz");
lineOfPeople.add("lisa");
lineOfPeople.add("hans");
println("There is a total of " + lineOfPeople.size() + " people waiting
in line.");

// who is in front of the queue
println("First person in line: " + lineOfPeople.peek());
println("There is a total of " + lineOfPeople.size() + " people waiting
in line.");

// remove one by one the top element from the stack
println("Customers checking out:");
println(lineOfPeople.remove());
println(lineOfPeople.remove());
println(lineOfPeople.remove());

println("There is a total of " + lineOfPeople.size() + " people waiting
in line.");
```

Auch für die Queue gibt es verschiedene Anwendungsszenarien:

- Telefonwarteschlange in einem Call Center;
- Druckaufträge für einen Drucker sind normalerweise in einer Warteschlange;
- Einchecken beim Flugzeug;
- die Tastenanschläge einer Tastatur.

Die *PriorityQueue* werden wir später noch ein paar mal wiedersehen.

Review

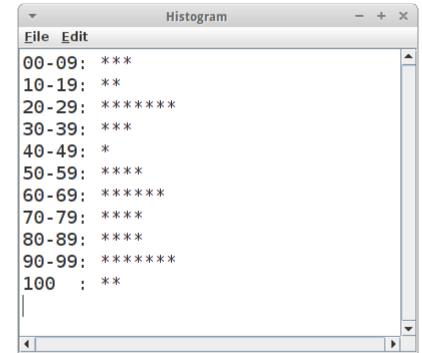
Obwohl es so aussieht wie wenn wir bisher nicht allzuviel gemacht hätten, soll das nicht darüber hinwegtäuschen, dass die Datenstrukturen *ArrayList*, *LinkedList* und *Stack* unser Brot und Butter sein werden. Auch die Queue werden wir ab und zu wiedersehen. Ähnlich wie das Ein-mal-eins, redet man da nicht groß drüber, aber man geht davon aus, dass es jeder wie aus dem "Effeff" kann.

Projekte

In den Projekten werden wir uns ganz kurz noch einmal das Array ansehen, dann ein paar Beispiele mit ArrayList und LinkedList sehen, und danach folgen dann Stack und Queue Beispiele. Das wohl interessanteste Projekt ist das ArithmeticExpression Projekt: mithilfe zweier Stacks implementieren wir einen Taschenrechner!

.

Histogram



Wir beginnen mit einer kleinen Anwendung die Arrays benutzt. Es geht darum ein Histogramm ausgeben zu lassen, als Beispieldaten dienen Punkte einer Klausur. Wir wissen, dass die Punkt zwischen 0 und 100 liegen, und wir möchten, dass es genau elf sogenannte *Bins* gibt, also Kategorien, in denen wir die Punkte kummulieren möchten. Da wir wissen, dass es genau elf gibt, nie mehr und nie weniger, bietet sich das Array als Datentyp an. Ansonsten müssten wir eine Liste nehmen.

```
private int[] histogramData = new int[11];
```

Das Befüllen der Bins ist relative einfach mit der Hilfe unseres Freunds Ganzzahldivision:

```
private void putScoreInHistogram(int score) {
    histogramData[score / 10]++;

}
```

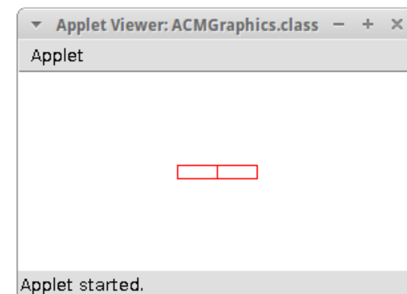
Die Daten selbst kommen aus der "Scores.txt" Datei, die wir Zeile für Zeile einlesen, und in Ganzzahlen umwandeln:

```
private void readData() {
    try {
        BufferedReader rd = new BufferedReader(new
FileReader("Scores.txt"));

        while (true) {
            String line = rd.readLine();
            if (line == null)
                break;
            int score = Integer.parseInt(line);
            putScoreInHistogram(score);
        }
        rd.close();
    } catch (Exception e) {
        e.printStackTrace();
    }

}
```

Wenn wir fertig sind, gehen wir das Array durch und geben es auf der Konsole aus. Wir könnten einfach die Zahlen ausgeben, viel hübscher sind aber kleine Sternchen (Asterisk nicht Asterix!):

```
private String convertToStars(int i) {
    String stars = "";
    for (int j = 0; j < i; j++) {
        stars += "*";
    }
    return stars;

}
```

ACMGraphics



Seit Beginn des letzten Semesters verwenden wir ja die ACM Graphikbibliothek *acm.jar*. Natürlich würde uns interessieren wie die wirklich funktioniert. Im Prinzip sind alle *Programs*, also *Program*, *ConsoleProgram* und *GraphicsProgram* eigentlich Unterklassen der Standard Java Klasse *Applet*. Das hat den angenehmen Nebeneffekt, dass alle unsere ACM Programme auch im Browser laufen (insofern der Browser Java unterstützt, was in letzter Zeit sehr wenige tun).

Ein Applet sieht im allgemeinen wie folgt aus:

```
public class ACMGraphics extends Applet {

    public void init() {
        ...
    }

    public void run() {
        ...
    }

    public void paint(Graphics g) {
        ...
    }

}
```

Es gibt also eine *init()* und eine *run()* Methode, und ja, das sind unsere *init()* und *run()* Methoden die wir schon die ganze Zeit verwenden. Außerdem gibt es auch noch die *paint()* Methode, auf die wir gleich kommen.

Was wir allerdings nicht sehen ist die *add()* Methode mit der wir unsere GRects und GOvals usw. hinzugefügt haben. Die Frage stellt sich wozu werden denn die GObjects hinzugefügt? Ja, richtig wenn das hier das Kapitel über Listen ist, dann wird es wohl einen Liste sein:

```
public class ACMGraphics extends Applet {

    private ArrayList<GRect> gObjects = new ArrayList<GRect>();

    private void add(GRect r) {
        gObjects.add(r);
    }

    ...

}
```

Das war der erste Teil. Jetzt stellt sich die Frage, wie zeichnet man denn die GObjects? Das passiert in der *paint()* Methode,

```
    public void paint(Graphics g) {
        for (GRect rect : gObjects) {
            rect.draw(g);
        }

    }
```

wo wir ein GObject nach dem anderen durchgehen und ihm sagen es soll sich doch selbst zeichnen. Wie funktioniert das? Dafür definieren wir eine Klasse *GRect*,

```
class GRect {
    private int x, y, w, h;

    public GRect(int x, int y, int w, int h) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
    }

    public void draw(Graphics g) {
        g.drawRect(x, y, w, h);
    }

}
```

und die benutzt die Standard Java Methode *drawRect()* der Graphics Klasse. Das war's. Mit einer kleinen Einschränkung: Animationen kann unsere einfache Version der ACM Library noch nicht, dafür müssen wir warten bis wir was von Multi-Threading gehört haben.

Homemade ArrayList

Wir wollen unsere eigene ArrayList Klasse schreiben, und sie soll (fast) alles können was die normale ArrayList Klasse von Java auch kann, also die folgenden Methoden haben:

- size()
- add(object)
- get(i)
- set(i, object)

Die *remove()* Methode haben wir absichtlich weggelassen, da die etwas komplizierter ist. Der Anfang unserer HomemadeArrayList sieht wie folgt aus:

```
public class HomemadeArrayList {

    private Object[] arr;
    private int capacity = 10;
    private int position = -1;

    public HomemadeArrayList() {
        arr = new Object[capacity];
    }

    public int size() {
        return position + 1;
    }

    ...
```

Die Daten unserer HomemadeArrayList werden in dem Array von Objekten, also Object[] arr, gespeichert. Das hat eine anfängliche Kapazität von 10, die wir im Constructor festlegen. D.h. erst mal können wir in unserer ArrayList nur zehn Objekte speichern.

Dann haben wir noch einen Positionspointer, *position*, der uns sagt, wieviel von dem Array schon belegt ist. Am Anfang ist es leer, deswegen -1. Wenn wir jetzt ein Element hinzufügen,

```
public void add(Object obj) {
    if (position < capacity-1) {
        position++;
        arr[position] = obj;
```

```
        } else {
            // we need to increase size of underlying array
        }

    }
```

müssen wir erst mal checken ob wir noch Platz haben, dann unseren Positionspointer um eins erhöhen, und natürlich nicht vergessen unser neues Element zu speichern. Wenn wir an die Grenzen unseres Arrays stoßen, haben wir Pech gehabt.

Das Lesen eines Objektes mit *get()* ist ganz einfach,

```
    public Object get(int i) {
        if (i >= 0 && i <= position) {
            return arr[i];
        }
        return null;

    }
```

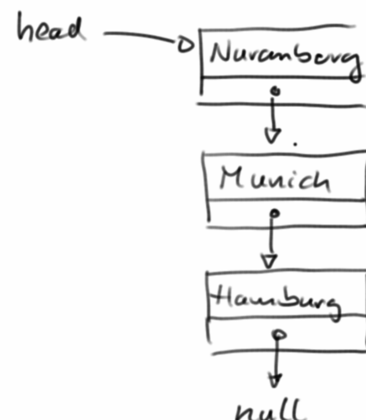
wir müssen lediglich darauf achten, dass die Werte von *i* in Ordnung sind. Schließlich fehlt nur noch die *set()* Methode,

```
    public void set(int i, Object obj) {
        if (i >= 0 && i <= position) {
            arr[i] = obj;
        }

    }
```

War gar nicht so schwer. Natürlich haben wir die schweren Sachen auch weg gelassen.

Challenge: Man könnte versuchen die remove() Methode zu implementieren. Und man könnte eine generische Klasse daraus machen.

Homemade LinkedList



Wir wollen jetzt unsere eigene LinkedList Klasse schreiben. Sie soll die folgenden Methoden haben:

- size()
- add(object)
- get(i)
- set(i, object)

Die *remove()* Methode haben wir auch wieder weggelassen. Bevor wir aber beginnen können, müssen wir uns kurz überlegen, wie eine LinkedList ihre Daten speichert. Das geht über die Verlinkung von Knoten. Ein Knoten ist eine einfache Klasse,

```
    class Node {
        public Node next;
        public Object obj;

    }
```

bestehend aus einem Objekt und einem Link zum nächsten Knoten. D.h. wir haben irgendwo einen ersten Knoten, *first*, und von dem aus hangeln wir uns von einem Knoten zum nächsten durch die ganze Liste, bis wir den Knoten haben, den wir suchten. Unsere *HomemadeLinkedList* Klasse hat also eine Referenz auf den ersten Knoten, und weiss noch wie groß sie ist:

```
public class HomemadeLinkedList {

    private Node first;
    private int size = 0;

    public HomemadeLinkedList() {
        // nothing to do
    }

    public int size() {
        return size;
    }

    ...
```

Wenn wir neue Elemente hinzufügen,

```
    public void add(Object obj) {
        if (first == null) {
            first = new Node();
            first.obj = obj;
            size++;
        } else {
            // find last node
            Node current = first;
            while (current.next != null) {
                current = current.next;
            }
            // we are at the end, add new one
            current.next = new Node();
            current.next.obj = obj;
            size++;
        }

    }
```

dann müssen wir als allererstes feststellen, ob überhaupt schon was in unserer Liste ist, also ob *first == null*. Falls das der Fall ist, müssen wir einen allerersten neuen Knoten anlegen. Falls nein, dann müssen wir den letzten Knoten finden, denn *add()* bedeutet ja, dass wir etwas ans Ende der Liste anfügen. Wenn wir dann etwas hinzugefügt haben, müssen wir natürlich die Größe, *size*, um eins erhöhen.
Das Lesen eines Objektes mit *get()* ist nicht ganz so einfach wie bei der ArrayList,

```
    public Object get(int i) {
        Node nde = getNode(i);
        if (nde != null) {
            return nde.obj;
        }
        return null;

    }
```

wir müssen erst einmal wissen, ob wir überhaupt ein Element *i* haben. Das macht die Methode *getNode()*:

```
    private Node getNode(int i) {
        int cnt = 0;
        Node current = first;
        while (current != null) {
            if (cnt == i) {
                return current;
            }
```

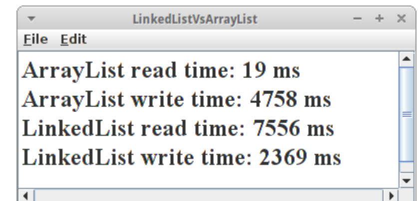
```
            current = current.next;
            cnt++;
        }
        return null;

    }
```

Im Prinzip geht diese Methode unsere Liste durch, einen Knoten nach dem anderen, beginnend bei dem ersten, *first*. Außerdem zählt sie mit, wieviele Knoten wir schon besucht haben, und wenn der Zähler *cnt* gleich dem gesuchten *i* ist, dann haben wir den Knoten gefunden.

Wenn wir einmal die Methode *getNode()* haben, dann ist auch das set() relativ problemlos:

```
    public void set(int i, Object obj) {
        Node nde = getNode(i);
        if (nde != null) {
            nde.obj = obj;
        }

    }
```

Auch hier kann man noch die *remove()* Methode implementieren und die Klasse generisch machen.



ArrayList vs LinkedList

Weiter oben haben wir uns die Frage gestellt, wann man eine *ArrayList* und wann eine *LinkedList* verwenden sollte. Die Antwort war: bei häufigem Lesen die ArrayList und bei häufigem Schreiben die LinkedList. Stimmt das aber auch?

Am besten wir prüfen das mal nach. Dazu messen wir einfach wie lange das Lesen bzw. Schreiben dauert. Wir schreiben also eine Methode *testRead(List<Integer> al)*, die wir einmal mit einer *ArrayList* und einmal mit einer *LinkedList* aufrufen:

```
println("ArrayList read time: " + testRead(new ArrayList<Integer>()) + "
ms");

println("LinkedList read time: " + testRead(new LinkedList<Integer>()) +
" ms");
```

(Wir sehen wie praktisch das List *Interface* ist.) Kommen wir zur Test Methode:

```
    private static long testRead(List<Integer> al) {
        // fill list with some dummy data:
        for (int i = 0; i < 1000000; i++) {
            al.add(42);
        }

        // start the read test:
        long start = System.currentTimeMillis();
        for (int i = 0; i < 100000; i++) {
            // read an element at a random position:
            int randomPos = (int) (100000.0 * Math.random());
            al.get(randomPos);
        }
        long end = System.currentTimeMillis();
        return (end - start);

    }
```

Im ersten Teil, befüllen wir die Liste mit einer Millionen Zahlen, immer die selbe. Danach beginnen wir mit unserem Test: wir lesen einhunderttausend Mal von einer zufälligen Position in der Liste. Die Zeit ermitteln wir mit der *System.currentTimeMillis()* Methode, die gibt uns die Zeit die seit dem 01.01.1970 vergangen ist in Millisekunden.

Das Ergebnis ist ziemlich eindeutig:

```
ArrayList read time: 19 ms

LinkedList read time: 7556 ms
```

Die ArrayList ist fast zehntausend mal schneller! Beim Schreiben ist der Unterschied nicht ganz so krass, da ist die LinkedList "nur" doppelt so schnell.

Vielleicht ist aufgefallen, dass wir hier eine *statische* Methode verwendet haben. Das ist ausnahmsweise mal o.k., weil wir Zeitmessungen machen, und statische Methoden etwas schneller sind. Das muss aber die Ausnahme bleiben, normalerweise dürfen wir *static* nie verwenden, auch wenn's schnell gehen soll! Der Grund ist, dass wir nicht mehr objekt-orientiert programmieren, wenn wir static verwenden.

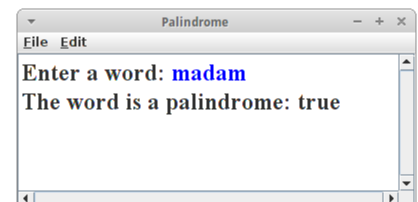
Fassen wir zusammen: Die ArrayList

+ hat sehr schnellen, direkten Zugriff auf Elemente, vor allem beim Lesen.
- Beim Einfügen und auch Entfernen von Elementen müssen alle nachfolgenden Elemente verschoben werden, und das führt dazu, dass diese Operation im Durchschnitt eher langsam ist.
- Stößt man an die interne Grenze des zugrundeliegenden Arrays, dann kostet es sehr viel Zeit ein einziges neues Element hinzuzufügen.

Bei der LinkedList ist

+ das Einfügen und Entfernen von Elementen eine einfache und schnelle Operation da nur ein Link neu gesetzt werden muss.
+ Da es kein zugrundeliegendes Array gibt, kann man auch nie an dessen Grenze stoßen.
- Um ein bestimmtes Element in der Liste zu finden, muss man u.U. durch die ganze Liste iterieren, da man keinen direkten Zugriff auf ein beliebiges Element hat. Jedes Element kennt nur sein Nachfolgeelement. Das dauert.

Palindrome



Wir wollen ein einfaches ConsoleProgram schreiben, das feststellt ob ein gegebenes Wort ein Palindrom ist, d.h. sowohl vorwärts als auch rückwärts das Gleiche bedeutet, wie z.B. "Madam".

```
public void run() {
    String text = readLine("Enter a word: ");
    println("The word is a palindrome: " + isPalindrome(text));

}
```

In der Methode *isPalindrome()* checken wir, ob das umgekehrte Wort mit dem ursprünglichen übereinstimmt:

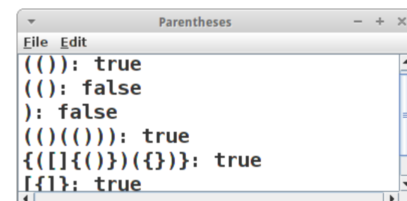
```
private boolean isPalindrome(String text) {
    String revers = reverseString(text);
    return revers.equals(text);

}
```

Kommen wir zur letzten Methode *reverseString()*: Es gibt natürlich mehrere Möglichkeiten einen String umzukehren, aber eine verwendet die Klasse Stack:

```
private String reverseString(String text) {
    Stack<Character> st = new Stack<Character>();
    for (int i = 0; i < text.length(); i++) {
        char c = text.charAt(i);
        st.push(c);
    }

    String revers = "";
    while (!st.isEmpty()) {
        char c = st.pop();
        revers = revers + c;
    }
    return revers;

}
```

Im ersten Teil definieren wir einen Stack für Characters, auf den wir einen Buchstaben nach dem anderen *pushen*. Im zweiten Teil holen wir einen Buchstaben nach dem anderen mittels *pop()* wieder vom Stack, bis dieser leer ist.

Das ist jetzt nicht die effektivste Art und Weise einen String umzukehren, aber es zeigt an einem einfachen Beispiel die Arbeitsweise der Klasse Stack.



Parentheses

Eine klassische Anwendung für die Stack Klasse ist es festzustellen, ob die Klammern in einem Ausdruck stimmen, also ob es für jede geöffnete Klammer auch wieder eine geschlossene gibt. Beispiele sind "(())" , "(()" , ")" und "(()(()))". Wir wollen also eine Methode *doParenthesesMatch()* schreiben, die feststellt ob alles stimmt:

```
public void run() {
    println("(()): " + doParenthesesMatch("(())"));
    println("((): " + doParenthesesMatch("(()"));
    println("): " + doParenthesesMatch(")"));
    println("(()(())): " + doParenthesesMatch("(()(()))"));
}
```

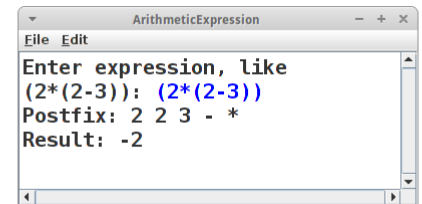
Dazu verwenden wir wieder einen Stack für Buchstaben:

```
private boolean doParenthesesMatch(String text) {
    Stack<Character> stack = new Stack<Character>();
    for (int i = 0; i < text.length(); i++) {
        char c = text.charAt(i);
        switch (c) {
        case '(':
            stack.push(c);
            break;
        case ')':
            if (!stack.isEmpty()) {
                stack.pop();
            } else {
                return false;
            }
            break;
        default:
            System.out.println("we should never get here!");
            break;
        }
    }
```

```
            // if parenthesis matched, the stack should be empty now:
            if (stack.isEmpty()) {
                return true;
            }
            return false;
        }
```

Wir lesen dann Zeichen für Zeichen ein. Wenn es eine offene Klammer ist, dann pushen wir die auf den Stack. Wenn es eine geschlossene Klammer ist, dann popen wir das oberste Element vom Stack. Hier müssen wir allerdings aufpassen, es könnte sein, dass gar nichts auf dem Stack ist. Das würde z.B. passieren wenn der Nutzer ")" eingibt. Sind wir dann durch, ist ganz einfach festzustellen, ob die gleiche Anzahl von offenen wie geschlossenen Klammern war: wenn nämlich der Stack leer ist, dann war alles in Ordnung.

Challenge: etwas komplizierter (aber nicht viel) wird es wenn man auch eckige und geschweifte Klammern zulässt: {([]{()})({})} und [{]}.



ArithmeticExpression

Kommen wir zu einer der wichtigsten Anwendung der Stack Klasse: der Auswertung arithmetischer Ausdrücke, wie z.B.,

```
1 + ( 2 + 3 * 4 + 3 ) / 2
```

Wir wollen uns aber der Einfachheit halber auf vollständig geklammerte arithmetische Ausdrücke, also z.B.

```
( ( 1 + ( 2 + 3 ) ) * ( ( 4 + 3 ) / 2 ) )
```

beschränken. Diese Form der Notation nennt man auch die *Infix* Notation (nicht mit der *Idefix* Notation zu verwechseln). Die *Infix* Notation ist zwar ganz praktisch für Menschen, aber für Computer taugt sie nicht viel. Computer lieben eher die *Postfix* Notation. Deswegen müssen wir aus *Infix* *Postfix* machen. Und dafür gibt es ganz einfache Regeln:

1. Wir lesen einen Token nach dem anderen;
2. ist der Token ein Operator, also '+', '-', '*' oder '/', dann schieben wir ihn auf den Stack;
3. ist er eine Ganzzahl, dann schreiben wir ihn in den *out* String;
4. ist er eine rechte Klammer ')', dann holen wir das oberste Element vom Stack und schreiben es in den *out* String;
5. ist er eine linke Klammer '(', dann machen wir gar nichts.

Das setzen wir einfach in der Methode *infixToPostfix()* um:

```
        public String infixToPostfix(String infix) {
            String out = "";
            Stack<String> stack = new Stack<String>();
            StringTokenizer st = new StringTokenizer(infix, "()+-*/ ", true);
            while (st.hasMoreTokens()) {
                String token = st.nextToken().trim();
                if (token.length() == 0) {
                    // do nothing
                } else if ("+-*/".contains(token)) {
                    stack.push(token);
                } else if (")".contains(token)) {
                    out += stack.pop() + " ";
                } else if ("(".contains(token)) {
                    // do nothing
                } else {
                    out += token + " ";
                }
            }
            return out;     }
```

Eine kleine Komplizierung kommt von den Leerzeichen, aus denen werden nach dem *trim()* leere Strings, die müssen wir explizit ignorieren. In der *Infix* Notation sieht unser arithmetischer Ausdruck so aus:

```
1 2 3 + + 4 3 + 2 / *
```

Man nennt das auch "reverse polish notation" (RPN). Wir tun uns damit schwer, aber Computer lieben diese Form, weil sie damit sofort weiterrechnen können. Auch hier gibt es wieder ganz einfache Regeln:

1. Wir lesen einen Token nach dem anderen;
2. ist der Token eine Ganzzahl, dann schieben wir sie auf den Stack;
3. ist es ein Operator, dann nehmen wir die beiden obersten Element vom Stack, führen den Operator mit diesen beiden Zahlen aus, und schieben das Ergebnis wieder auf den Stack;
4. am Ende dürfte nur noch ein Element auf dem Stack sein, und das ist das Ergebnis.

Das setzen wir in der Methode *evaluatePostfix()* um:

```
public int evaluatePostfix(String postfix) {
    Stack<Integer> stack = new Stack<Integer>();
    StringTokenizer st = new StringTokenizer(postfix, "+-*/ ", true);
    while (st.hasMoreTokens()) {
        String token = st.nextToken().trim();
        if (token.length() == 0) {
            // do nothing
        } else if ("+-*/".contains(token)) {
            int y = stack.pop();
            int x = stack.pop();
            if (token.equalsIgnoreCase("+")) {
                stack.push(x + y);
            } else if (token.equalsIgnoreCase("-")) {
                stack.push(x - y);
            } else if (token.equalsIgnoreCase("*")) {
                stack.push(x * y);
            } else if (token.equalsIgnoreCase("/")) {
                stack.push(x / y);
            }
        } else {
            stack.push(Integer.parseInt(token));
        }
    }
    return stack.pop();

}
```

Wenn wir das auf den Ausdruck oben anwenden, sollte 18 als Ergebnis herauskommen.

An dieser Stelle sei aber noch einmal auf die Einschränkung hingewiesen: unser Algorithmus funktioniert nur für vollständig geklammerte arithmetische Ausdrücke. Unser Algorithmus kennt auch kein Punkt-vor-Strich. D.h., wenn wir ihm unseren Ausdruck aus dem ersten Semester

```
( 1 + 3 * 5 / 2 )
```

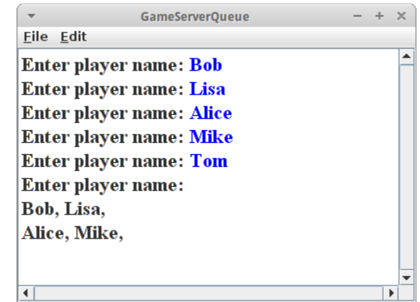
vorsetzen würden, käme da etwas falsches heraus. Allerdings wenn wir ihm

```
( 1 + ( ( 3 * 5 ) / 2 ) )
```

vorsetzen, dann macht er alles richtig. Es ist nicht allzu schwer unseren Algorithmus so zu modifizieren, dass er auch noch Punkt-vor-Strich kann, aber didaktisch ist er dann nicht mehr ganz so einfach. Wen es interessiert kann bei [1] und [2] nachlesen.

Wir sollten kurz innehalten, um uns der Tragweite bewusst zu werden was wir gerade geschafft haben: mit Hilfe des Stacks haben wir eine Rechenmaschine gebaut. Es stellt sich heraus, dass die Java Virtual Machine auch eine Stack-Maschine ist, d.h., was wir hier gemacht haben, ist fast schon eine Virtual Machine programmiert - fast.

GameServerQueue



Multiplayer Games benötigen eine gerechte Möglichkeit Spieler zu Teams zusammenzufassen. Dafür verwendet man am besten eine Queue. Nehmen wir an wir wollen Teams mit jeweils zwei Spielern bilden.

```
public class GameServerQueue extends ConsoleProgram
{
    private final int NR_OF_PLAYERS_PER_TEAM = 2;
    private Queue<String> playerQueue = new
LinkedList<String>();

    public void run() {
        ...
    }

}
```

Als erstes fügen wir ein paar Spieler in die Schlange ein, z.B. fünf verschiedene Namen:

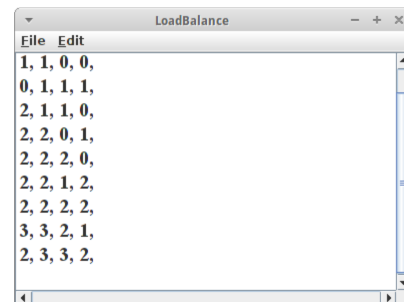
```
    private void fillQueueWithNames() {
        while (true) {
            String name = readLine("Enter player name: ");
            if (name.length() == 0)
                break;
            playerQueue.add(name);
        }

    }
```

Diese werden in der Schlange gespeichert in der Reihenfolge in der sie eingegeben wurden. Dann ist es an der Zeit Paare zu bilden, also jeweils zwei Spieler aus der Queue zu entfernen:

```
    private void removePlayers() {
        if (playerQueue.size() >= NR_OF_PLAYERS_PER_TEAM) {
            for (int i = 0; i < NR_OF_PLAYERS_PER_TEAM; i++) {
                print(playerQueue.remove() + ", ");
            }
            println();
        }

    }
```

Wie wir sehen, passiert das in der richtigen Reihenfolge. Außerdem bleibt der fünfte Spieler in der Schlange, solange bis ein neuer Spieler hinzukommt, denn mit einem Spieler kann man kein Paar bilden.

LoadBalance



LoadBalance ist eine typische Queue Anwendung. Wir haben z.B. vier Server, und wir möchten, dass die Last gleichmässig auf die Server verteilt wird. Es gibt verschiedene Möglichkeiten das zu tun. Bei *RoundRobin* z.B. wird die Last einfach der Reihe nach verteilt. Eine andere berücksichtigt die Last des Servers. Soll heißen, der Server mit der geringsten Last, soll den neuesten Task zugewiesen bekommen.

Wir schreiben ein einfaches ConsoleProgram, in dem wir die *serverQueues* als Array definieren, und in der *init()* Methode initialisieren:

```
public class LoadBalance extends ConsoleProgram {
    private final int NR_OF_SERVERS = 4;
    private Queue<String>[] serverQueues;
    private int currentTaskNr = 0;

    public void init() {
        serverQueues = new Queue[NR_OF_SERVERS];
        for (int i = 0; i < serverQueues.length; i++) {
            serverQueues[i] = new LinkedList<String>();
        }
    }

    ...
```

In der *run()* Methode,

```
    public void run() {
        while (true) {
            addNewServerTask();
            addNewServerTask();
            removeSomeRandomTask();
            printServerLoad();
            pause(1000);
        }

    }
```

fügen wir jeweils zwei neue Tasks hinzu, und entfernen einen zufälligen alten. Dazwischen lassen wir uns immer die Serverlast ausgeben:

```
    private void printServerLoad() {
        for (int i = 0; i < serverQueues.length; i++) {
            print(serverQueues[i].size() + ", ");
        }
        println();

    }
```

das ist einfach die Größe jeder der vier Queues. Das Entfernen ist ganz einfach zufällig:

```
    private void removeSomeRandomTask() {
        int serverNr = (int) (NR_OF_SERVERS * Math.random());
        if (serverQueues[serverNr].size() > 0) {
            serverQueues[serverNr].remove();
        }

    }
```

Bevor wir neue Tasks hinzufügen können, müssen wir erst einmal wissen welcher der Server der mit der geringsten Last ist:

```
    private int findServerWithLowestLoad() {
        int serverNr = 0;
        int min = serverQueues[serverNr].size();
        for (int i = 1; i < serverQueues.length; i++) {
            if (min > serverQueues[i].size()) {
                min = serverQueues[i].size();
                serverNr = i;
            }
        }
        return serverNr;

    }
```

Wir gehen einfach alle Server durch und finden denjenigen, dessen Queue am wenigsten Tasks enthält. Danach geben wir diesem einen neuen Task:

```
    private void addNewServerTask() {
        int serverNr = findServerWithLowestLoad();
        currentTaskNr++;
        serverQueues[serverNr].add("Task Nr." + currentTaskNr);

    }
```

Und das war's schon.

Challenges

RandomArt

Mit dem Beispiel ArithmeticExpression kann man auch hübsche, zufällige Graphiken erzeugen. Der Code ist fast identisch, lediglich die Regeln ändern sich ein wenig.

Bevor wir aber beginnen, wollen wir erst einmal eine interessante Beobachtung machen: der Sinus und Cosinus einer beliebigen Zahl ergibt immer einen Wert zwischen -1 und +1. Wenn wir zwei Zahlen die zwischen -1 und +1 sind miteinander multiplizieren, dann ergibt das immer eine Zahl die zwischen -1 und +1 liegt. Und wenn wir den Durchschnitt zweier Zahlen bilden die zwischen -1 und +1 sind, dann kommt da auch wieder eine Zahl zwischen -1 und +1 raus. Diese Tatsache lässt sich nutzen um hübsche Graphiken zu erzeugen.

Für unser RandomArt Projekt wollen wir also die vier mathematischen Funktionen *sine*, *cosine*, *multiplication* und *average* verwenden. Dabei beginnen wir mit zwei Variablen *x* und *y* mit denen wir dann folgendes ausrechnen können:

```
z = s( x )      // sine
z = c( x )      // cosine
z = ( x * y )   // multiply x by y

z = ( x , y )   // average x and y
```

Wenn *x* und *y* zwischen -1 und +1 liegen, dann wird auch *z* wieder zwischen -1 und +1 liegen. Wir könnten also *z* wieder in eine der vier Funktionen stecken, und das *rekursiv* immer wieder tun. Das ist was die Funktion *createMathFunction()* macht:

```
private String createMathFunction() {
    String s = "";
    int key = (int) (Math.random() * 6);
    switch (key) {
    case 0:
        s = "x";
        break;
    case 1:
        s = "y";
        break;
    case 2:
        // cosinus
        s = "c(" + createMathFunction() + ")";
        break;
    case 3:
        // sinus
        s = "s(" + createMathFunction() + ")";
        break;
    case 4:
        // multiplication
        s = "(" + createMathFunction() + "*" + createMathFunction() +
")";
        break;
    case 5:
        // average:
        s = "(" + createMathFunction() + "," + createMathFunction() +
")";
        break;
    }
    return s;

}
```

Da hier der Zufall mitspielt kann man nicht genau sagen was rauskommt, aber ein Resultat dieser Methode könnte wie folgt aussehen:

```
c(c(s(c((c(s(s(x))),(y*y))))))
```

Das erinnert sehr an die *infix* Notation der ArithmeticExpression, und das ist es auch. Es stellt sich heraus, dass zu kurze Ausdrücke nach nichts aussehen und zu lange Ausdrücke benötigen zu viel Zeit zum darstellen, deswegen suchen wir so lange bis wir eine *infix* Notation finden die irgendwo zwischen 50 und 200 Zeichen lang ist:

```
private String findGoodPostfix() {
    String infix = "";
    while (true) {
        infix = createMathFunction();
        if ((infix.length() > 50) && (infix.length() < 200))
            break;
    }
    String postfix = convertFromInfixToPostfix(infix);
    return postfix;

}
```

Aus dieser *infix* Notation machen wir dann genauso wie bei ArithmeticExpression eine postfix Notation in der Methode *convertFromInfixToPostfix()*. Die folgt auch den gleichen Regeln:

1. Wir lesen einen Token nach dem anderen;
2. ist der Token ein Operator, also 's', 'c', '*' oder ',', dann schieben wir ihn auf den Stack;
3. ist es ein 'x' oder 'y' ist, dann schreiben wir es in den *out* String;

4. ist es eine rechte Klammer ')', dann holen wir das oberste Element vom Stack und schreiben es in den *out* String;
5. ist es eine linke Klammer '(', dann machen wir gar nichts.

Analog zum ArithmeticExpression Beispiel benötigen wir noch die *evaluate()* Methode. Auch hier sind die Regeln fast identisch:

1. Wir lesen einen Token nach dem anderen;
2. ist der Token ein 'x' oder 'y', dann schieben wir ihn auf den Stack;
3. ist es ein binärer Operator, also entweder '*' oder ',', dann nimm die beiden obersten Elemente vom Stack, führe den Operator mit diesen beiden Zahlen aus (also Multiplikation oder Durchschnitt bilden), und schiebe das Ergebnis wieder auf den Stack;
4. ist es ein unärer Operator, also entweder 's' oder 'c', dann nimm das oberste Elemente vom Stack, führe die Operation aus (also nimm entweder den Sinus oder Cosinus dieser Zahl), und schiebe das Ergebnis wieder auf den Stack;
5. am Ende dürfte nur noch ein Element auf dem Stack sein, und das ist das Ergebnis.

Jetzt benötigen wir lediglich eine Methode um unsere Kunstwerke zu zeichnen. Wir gehen sowohl für *x* als auch für *y* alle Werte zwischen -1 und +1 durch, und rufen für jeden die *evaluate()* Methode auf:

```
private void drawArt(String postfix) {
    double step = 2.0 / SIZE;
    long startTime = System.currentTimeMillis();
    for (double x = -1.0; x < 1.0; x += step) {
        for (double y = -1.0; y < 1.0; y += step) {
            double col = evaluate(postfix, x, y);
            setPixel(x, y, col);
        }
    }

}
```

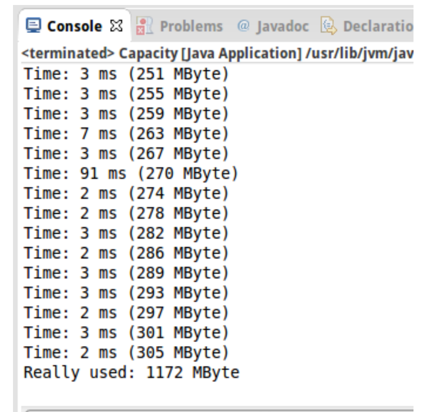
Die gibt uns einen Wert zwischen -1 und +1 zurück, den wir dann auf einen Grauwert zwischen 0 und 255 skalieren, und zeichnen:

```
private void setPixel(double x, double y, double col) {
    int i = (int) ((x + 1.0) * SIZE) / 2;
    int j = (int) ((y + 1.0) * SIZE) / 2;
    GLine r = new GLine(0, 0, 0, 1);
    int color = (int) ((col + 1.0) * 255) / 2;
    r.setColor(new Color(color, color, color));
    add(r, i, j);

}
```

Natürlich müssen wir auch noch die x- und y-Positionen skalieren. Die Idee für dieses Projekt stammt von Christopher Stone [3], welches wiederum von Andrej Bauer inspiriert wurde [4].

Capacity*

Eine ArrayList verwendet ja ein internes Array. Die Frage ist, können wir das beweisen? Ja, können wir. Denn jedes Mal wenn das interne Array voll ist, muss ein neues angelegt werden, und das alte muss rüberkopiert werden. Und das dauert. Wie man in dem Screenshot rechts sehen kann, dauert es normalerweise zwischen 2 und 7 ms um eine Millionen Zahlen in die ArrayList einzufügen. Ab und zu aber dauert es länger, hier sind es z.B. 91 ms, das ist zehnmal länger.



Um das zu messen benötigen wir erst einmal eine Methode die eine Millionen Zahlen (also grob 4 MByte) in die Liste einfügt:

```
    private static void addSomeEntries(List<Integer> al) {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 1000000; i++) {
            al.add( 42 );
        }
        long end = System.currentTimeMillis();
        System.out.println("Time: " + (end-start) + " ms
("+al.size()*4/ONE_MEGABYTE+" MByte)");

    }
```

Diese Methode führen wir dann einfach 80 mal aus und beobachten was passiert:

```
    public static void main(String[] args) {
        List<Integer> al = new ArrayList<Integer>();
        //List<Integer> al = new LinkedList<Integer>();
        long startMem = usedMemory();
        for (int i = 0; i < 80; i++) {
            addSomeEntries(al);
        }
        long endMem = usedMemory();

        System.out.println("Really used: "+(endMem-
startMem)/ONE_MEGABYTE+" MByte");

    }
```

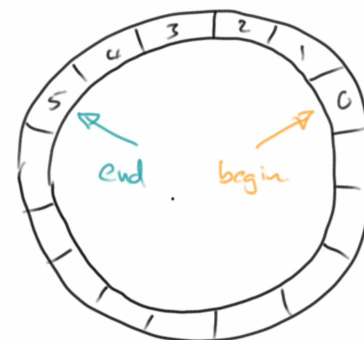
Nur so Interesse halber wollen wir auch mal sehen wieviel Speicher unser Programm wirklich benötigt, und das geht mit *usedMemory()*:

```
    private static long usedMemory() {
        Runtime runtime = Runtime.getRuntime();
        return runtime.totalMemory() - runtime.freeMemory();

    }
```

Die Methode ist etwas mit Vorsicht zu geniesen, da sie den freien Speicher des gesamten Betriebssystems wiedergibt. Wenn wir also gleichzeitig alle möglichen anderen Sachen auf unserem Rechner laufen lassen, dann wird das nicht funktionieren.

Wenn wir ein bisschen mit dem Programm spielen, und auch mal LinkedLists anstelle von ArrayLists verwenden stellen wir einige interessante Dinge fest:

- die meisten Inserts gehen bei der ArrayList sehr schnell, aber hin und wieder dauert es länger, bei einer richtigen großen Liste kann es schon mal ne Sekunde oder länger dauern;
- irgendwie scheint Java nicht besonders effektiv mit dem Speicher umzugehen: wir hätten erwartet, dass lediglich 300 MByte verbraucht werden, in Wirklichkeit scheint unser kleines Programm aber etwas mehr als 1 GByte verbraucht zu haben;
- eine LinkedList sollte eigentlich nie das Problem mit der Capacity haben, aber auch hier stellen wir fest, dass es ab und zu ewig dauert ein einziges neues Element einzufügen. Verstehen tue ich das auch nicht.



Circular Buffer

Ein Ringpuffer ist eine Queue-artige Datenstruktur, also eine FIFO Datenstruktur, die allerdings eine feste vorgegeben Größe hat [5]. Dies wird sehr häufig für Log-Dateien, für Round-Robin im Load-Balancing oder für die Übertragung von Daten zwischen asynchronen Prozessen verwendet.

In dieser Übung sollen wir eine generische Klasse namens *CircularBuffer* schreiben, die einen Ringpuffer mit folgender Funktionalität umsetzt:

- **isEmpty():** gibt *true* zurück, falls der Puffer leer ist;
- **isFull():** gibt *true* zurück, falls der Puffer voll ist;
- **enqueue():** fügt eine neues Element am Ende ein, falls der Puffer allerdings voll ist, gibt die Methode *false* zurück, ansonsten *true*;
- **dequeue():** gibt das Element am Anfang zurück, oder *null*, falls der Puffer leer ist.

Am einfachsten verwendet man ein Array um das zu implementieren.

Research

In diesem Kapitel gibt es nicht ganz so viel zu erforschen.

List of Data Structures

Um eine Vorstellung davon zu bekommen, wie viele Datenstrukturen es da draußen gibt, werfen wir einen Blick auf die Sammlung von Datenstrukturen, die in der Wikipedia gelistet werden [6].

Double-Ended Queue

Es gibt Anwendungen bei denen bräuchte man eine Queue, bei der man sowohl am Anfang als auch am Ende etwas einfügen könnte, und genauso auch an beiden Enden etwas entfernen könnte. Dafür gibt es in Java die Klasse *ArrayDeque*. Finden Sie ein oder zwei mögliche Anwendungsfälle für eine derartige Datenstruktur.

Fragen

1. Erklären Sie den Unterschied zwischen einer LinkedList und einer ArrayList.

2. Geben Sie je eine Beispielanwendung für einen Stack und eine Queue.

3. Welchen Datenstruktur verwenden Sie für Matching Parentheses?

4. Vergleichen Sie das einfache Java-Array (int []) mit der ArrayList-Klasse. Nennen Sie je drei Unterschiede.

5. Nennen Sie die Namen von drei Methoden der Stack-Klasse.

6. Beschreiben Sie wie der Stack nach folgenden Operation aussieht:
   push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop()

Referenzen

Eine sehr schöne Einführung zu den Stack und Queue Datenstrukturen gibt das Buch von Robert Sedgewick and Kevin Wayne [7].

[1] Introduction to Programming in Java, Robert Sedgewick and Kevin Wayne, http://introcs.cs.princeton.edu/java/43stack/Evaluate.java.html

[2] Victor S.Adamchik, Computer Science - 121, Fall 2009, https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html

[3] Random Art, Christopher A. Stone, nifty.stanford.edu/2009/stone-random-art/

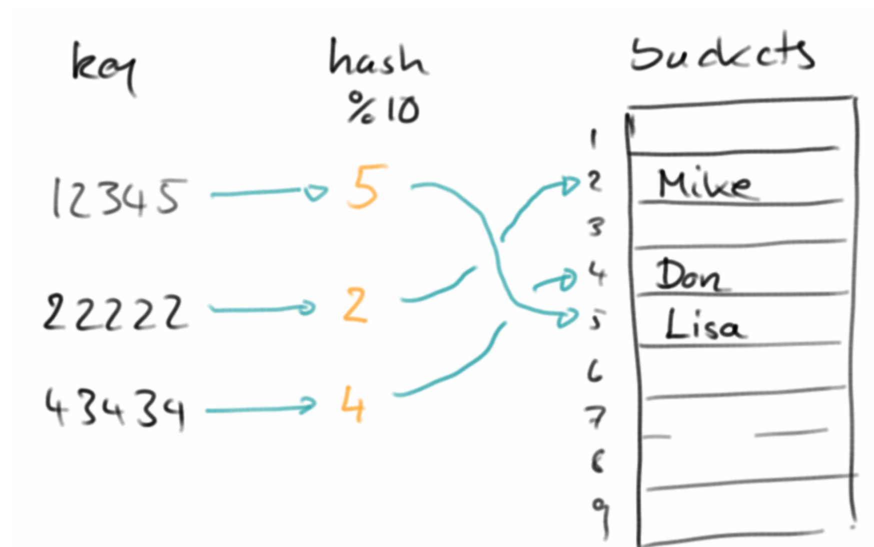
[4] Random Art, Andrej Bauer, www.random-art.org

[5] Wikipedia: Circular buffer, http://en.wikipedia.org/wiki/Circular_buffer

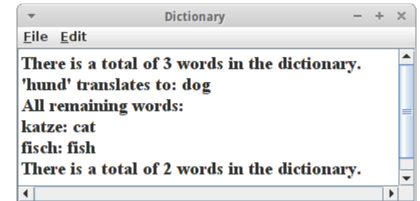
[6] List of data structures, http://en.wikipedia.org/wiki/List_of_data_structures

[7] Introduction to Programming in Java, by Robert Sedgewick and Kevin Wayne, at http://www.cs.princeton.edu/introcs/43stack/

Containers: Maps & Sets



Nach den sequentiellen Containern wollen wir uns nun mit den assoziativen Containern beschäftigen. Beispiele für assoziative Container sind Maps und Sets. Assoziative Container haben keine Reihenfolge, dafür haben sie aber eine Beziehung, in der Regel durch Schlüssel-Wert Paare. Assoziative Container sind extrem schnell, sowohl beim Lesen als auch beim Schreiben.



Maps

Eine Map ist eine Sammlung von Schlüssel-Wert Paaren (key-value pairs), die manchmal auch als Wörterbuch (dictionary) bezeichnet wird. Sie hat die Eigenschaft, dass mit jedem Schlüssel genau ein Wert assoziiert ist. Und die Schlüssel sind eindeutig, es kann also keine zwei Schlüssel mit dem gleichen Wert geben. Die grundlegenden Methoden, die von einer Map unterstützt werden, sind:

- **size():** gibt die Anzahl der Schlüssel (und/oder Werte) in einer Map;
- **put(key, value):** fügt ein neues Schlüssel-Wert Paar in die Map ein;
- **get(key):** gibt den Wert zurück, der unter diesem Schlüssel gespeichert ist;
- **remove(key):** entfernt den entsprechenden Schlüssel und damit natürlich auch den dazugehörigen Wert;
- **containsKey(key):** stellt fest ob der gegeben Schlüssel in der Map vorhanden ist. Ist extrem schnell.

In Java gibt es mehrere Implementierungen des Map Interfaces: die wichtigsten sind die *HashMap* und die *TreeMap*.

Sehen wir uns einfach mal ein Beispiel an. Wir verwenden eine *HashMap* für ein deutsch-englisches Wörterbuch:

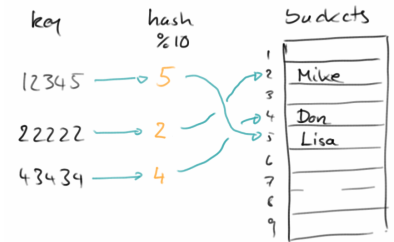
```
// init map
Map<String, String> dictionary = new HashMap<String, String>();

// add words
dictionary.put("hund", "dog");
dictionary.put("katze", "cat");
dictionary.put("fisch", "fish");
println("There is a total of " + dictionary.size()
        + " words in the dictionary.");

// translate a word
println("'hund' translates to: " + dictionary.get("hund"));

// remove a word
dictionary.remove("hund");

// list all remaining words
println("All remaining words:");
for (String word : dictionary.keySet()) {
    println(word + ": " + dictionary.get(word));

}
```

Typische Anwendung für Maps sind z.B.,

- ein Wörterbuch, das ein deutsches Wort auf ein englisches Wort abbildet;
- ein Telefonbuch, das einen Namen auf eine Telefonnummer abbildet;
- ein Thesaurus, der ein Wort auf seine Synonyme abbildet.

Mit Maps kann man aber auch alles machen, was man auch mit Listen machen kann. Da Maps aber um einiges schneller sind, sollten wir eigentlich nur noch Maps verwenden.

HashMap



Die Map die wir am häufigsten verwenden werden ist die *HashMap* (auf Deutsch Streuwerttabelle). Die zugrundeliegende Datenstruktur einer HashMap ist ein ganz normales Array. Das wird ähnlich wie bei der ArrayList am Anfang mit einer gewissen Anfangsgröße, der Capacity, angelegt. Was allerdings ganz anders ist, wie auf das Array zugegriffen wird. Hier verwendet die HashMap eine sogenannt *Hash-Funktion*.

Um zu verstehen wie das funktioniert, nehmen wir mal an wir möchten ein umgekehrtes Telefonbuch implementieren, d.h., wir sehen die Nummer einer Person die uns anruft auf dem Display, und wir möchten wissen wer das ist:

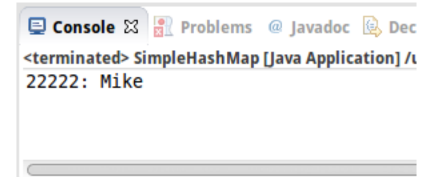
```
12345  ->  Lisa
22222  ->  Mike

43434  ->  Don
```

Wenn jetzt die Capacity unseres zugrundeliegenden Arrays 10 ist, dann könnte man folgende Funktion als *Hash-Funktion* verwenden,

```
int index = number % 10;
```

und *index* wäre dann die Position im Array unter der wir den Namen speichern würden, also Lisa an Index 5, Mike an Index 2 und Don an Index 4.

Wir können das auch ganz einfach in Code umsetzen: Wir definieren eine Klasse *SimpleHashMap* wie folgt:




```
public class SimpleHashMap {

    private String[] data;
    private int capacity = 10;

    public SimpleHashMap() {
        data = new String[capacity];

    }
```

D.h. wir haben ein String-Array angelegt für zehn Strings. Zusätzlich definieren wir unsere Hash-Funktion,

```
    private int hashCode(int key) {
        return key % capacity;

    }
```

Was noch fehlt sind die *put()* und *get()* Methode, die sind trivial:

```
    private void put(int key, String value) {
        data[hashCode(key)] = value;
    }

    private String get(int key) {
        return data[hashCode(key)];
    }

}
```

Das war's schon. Wir können jetzt unsere *SimpleHashMap* wie jede normal Map verwenden:

```
SimpleHashMap reversePhoneBook = new SimpleHashMap();

reversePhoneBook.put(12345, "Lisa");
reversePhoneBook.put(22222, "Mike");
reversePhoneBook.put(43434, "Don");


println("22222: " + reversePhoneBook.get(22222));
```

Der Vorteil der HashMap ist, dass sie sehr schnell beim Lesen und sehr schnell beim Schreiben ist, also das Beste was uns passieren kann.

Allerdings hat auch die HashMap ein paar Einschränkungen:

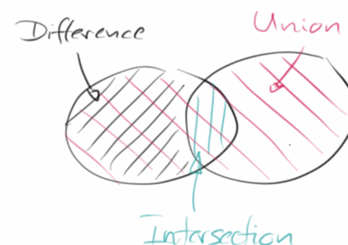
1. Wenn wir mehr als 10 Elemente in unserer *SimpleHashMap* schreiben wollen, dann geht uns der Platz aus. Das ist aber das gleiche Problem bei der ArrayList. Wir lösen es, in dem wir einfach ein größeres Array nehmen.
2. Es kann zu sogenannten Collisions kommen, das ist wenn zwei verschiedene Nummern auf den gleichen Index zeigen würden. Z.B. bei unserer einfachen Hash-Funktion oben, würden die beiden Nummer 12345 und 55555 beide auf den Index 5 gemappt werden, wir würden also versuchen zwei Einträge an der gleichen Stelle zu speichern. Zum Einen versucht man Kollisionen durch die Wahl guter Hash-Funktionen zu minimieren. Kommen sie trotzdem vor, dann kann man das auch lösen, indem man z.B. am Index 5 einfach eine Liste speichert, und dort dann beide Werte reinschreibt.
3. HashMaps haben keine Ordnung: d.h. wenn wir über die Elemente der Map iterieren, ist die Reihenfolge nicht vorhersehbar. Bei einer HashMap ist das einfach so. Aber es gibt die Klasse *LinkedHashMap* bei der die Reihenfolge in der die Elemente eingefügt wurden beibehalten wird, und es gibt die *TreeMap*, die ihre Elemente sortiert. Beide sind aber nicht ganz so schnell wie die normale HashMap.

Zum Schluss noch eine kleine Anmerkung bzgl. guter Hash-Funktionen: Hier kann man sich von Java selbst inspirieren lassen, im Source Code der OpenJDK findet man die Implementierung der *hashCode()* Funktion der Klasse String:

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;

}
```

Der Algorithmus erinnert etwas an den ISBN Algorithmus. Es sei angemerkt, wenn wir vorhaben, unsere selbst geschriebenen Klassen in einer HashMap zu speichern dann sollten wir unbedingt die *hashCode()* Methode unserer Klasse überschreiben.

Sets



Kommen wir zur zweiten Datenstruktur in diesem Kapitel, den *Sets* (Mengen auf Deutsch). Sets sind amputierte Maps, ihnen fehlen die Werte. Ein Set ist also keine Sammlung von Schlüsselwertpaaren, sondern eine Sammlung von nur Schlüsseln. Das hört sich jetzt schlimm an und scheint auch nutzlos zu sein, ist es aber nicht, ganz im Gegenteil.

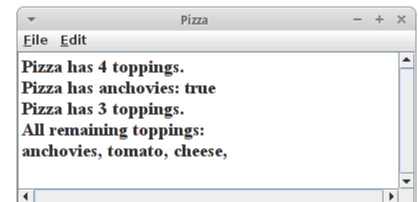
Die Schlüssel bei den Sets sind wie bei den Maps auch eindeutig, es kann also keine zwei Schlüssel mit dem gleichen Wert geben. Das ist auch schon eine der ersten praktischen Eigenschaften wofür man Sets verwenden kann: zum Entfernen von Duplikaten. Sets verwendet man auch wenn man feststellen möchte ob ein Element in einem Set ist oder nicht. Das geht extrem schnell. Und wenn man zwei Sets hat, kann man diese Zusammenfügen (Union), die Schnittmenge der beiden bilden (Intersection), oder die eine von der anderen abziehen (Difference). Man kann auch feststellen ob zwei Sets gleich sind, oder ob ein Set eine Untermenge des anderen ist. All diese Operationen sind extrem schnell.

Welche Methoden braucht ein Set? Die einfacheren sind,

- **size():** gibt die Größe des Sets zurück;
- **add(key):** fügt ein neues Element ins Set ein;
- **remove(key):** entfernt das Element *key* aus dem Set;
- **contains(key):** stellt fest ob das Element *key* im Set ist. Extrem schnell.

In Java verwenden wir normalerweise das *HashSet*, wenn's sortiert sein soll auch mal das *TreeSet*.

Um zu sehen, wie dies in der realen Welt verwendet wird, schauen wir uns ein Beispiel an:



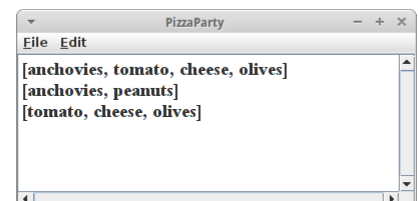

```
// init set
Set<String> pizza = new HashSet<String>();

// add entries
pizza.add("tomato");
pizza.add("olives");
pizza.add("cheese");
pizza.add("anchovies");
println("Pizza has " + pizza.size() + " toppings.");

println("Pizza has anchovies: " + pizza.contains("anchovies"));

// remove one topping
pizza.remove("olives");
println("Pizza has " + pizza.size() + " toppings.");

// list all remaining toppings
println("All remaining toppings:");
for (String topping : pizza) {
    print(topping + ", ");

}
```

Das ist zwar alles sehr nett, aber nicht wirklich nützlich. Was wirklich den Unterschied macht sind die erweiterten Funktionen:

- **addAll(set):** fügt eine ganzes Set diesem hinzu, wird auch als *Union* bezeichnet;
- **retainAll(set):** behalte nur die Elemente die in beiden Sets sind, wird auch *Intersection* genannt;
- **removeAll(set):** entferne alle Elemente die in dem anderen Set sind, bezeichnet man auch als *Subtraction*;
- **equals(set):** stellt fest ob die zwei Sets identisch sind;
- **containsAll(set):** stelle fest ob das Set *set* eine Untermenge ist.

Alle diese Operationen sind sehr schnell!

Um zu sehen, wie wir diese Methoden verwenden, kehren wir noch einmal zu unserem Pizzabeispiel zurück:




```
// init set
Set<String> pizza = initPizza();
println(pizza);

Set<String> allergic = new HashSet<String>();
```

```
allergic.add("anchovies");
allergic.add("peanuts");
println(allergic);

pizza.removeAll(allergic);
//pizza.addAll(allergic);
//pizza.retainAll(allergic);

println(pizza);
```

Wie praktisch das ist sieht man erst, wenn man sich nur einmal ansatzweise überlegt wie man das denn mit Arrays umsetzen wollte. Das Set ist unberechtigter Weise eine etwas vernachlässigte Datenstruktur. Es lohnt sich sie kennenzulernen, denn sie ist bei weitem das schnellste was es so an Datenstrukturen gibt.

Iterator

Die Existenz des Iterators haben wir bisher unterschlagen, obwohl wir ihn schon zweimal verwendet haben. Als wir im letzten Kapitel alle Städte unserer Rundreise auflisten wollten, hätten wir einen ganz normalen for-Loop verwenden können:

```
// list all cities via normal for loop:
for (int i = 0; i < cities.size(); i++) {
    String city = cities.get(i);
    print(city + ", ");

}
```

Anstelle haben wir aber den *for-each* Loop verwendet:

```
// list all cities via for-each loop:
for (String city : cities) {
    print(city + ", ");

}
```

Man liest das so: "für jede *city* aus der Liste *cities* tue das folgende...". Es ist super-praktisch und erspart einem viel unnötige Schreibarbeit.

Woher kommt das aber? Der for-each Loop benutzt implizit den *Iterator*:

```
// list all cities via iterator:
Iterator iter = cities.iterator();
while (iter.hasNext()) {
    String city = (String) iter.next();
    print(city + ", ");

}
```

Sowohl Listen, als auch Maps und Sets stellen einen Iterator über die Methode *iterator()* zu Verfügung. Dieser erlaubt es durch die Liste oder Map durchzuiterieren, d.h. beginnend vom ersten Element, eine Element nach dem anderen aufzulisten. Dazu gibt es die zwei Methoden:

- **hasNext():** gibt true zurück, falls es noch weitere Elemente gibt und
- **next():** gibt das nächste Element zurück.

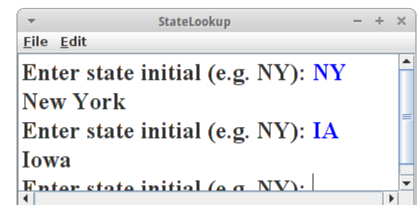
Die beiden erinnern eine bisschen an die *hasMoreTokens()* und *nextToken()* Methoden des StringTokenizer, und effektiv sind sie nichts anderes. Bei Listen haben wir die Wahl, ob wir mit einem klassischen for Loop oder dem for-each Loop iterieren wollen. Bei Maps oder Sets können wir nur mit dem for-each Loop iterieren, denn es gibt keine *get(i)* Methode für Map oder Set.

Review

Maps und Sets sind die wichtigsten Datenstrukturen überhaupt. Sie sind schnell, effizient und einfach zu benutzen. Man kann grob sagen, dass in 80% aller Fälle in denen wir nach einer Container Datenstruktur suchen, die Map die richtige Wahl ist. Dann kommen Sets, und eigentlich kann man davon ausgehen, wenn man eine Liste nimmt, dass man die falsche Datenstruktur gewählt hat. Bei Arrays sowieso, die verwenden nur Programmierer aus dem letzten Jahrhundert. Natürlich sind die Container Klassen für sich alleine schon cool genug, richtig interessant wird es aber erst wenn man anfängt sie zu kombinieren.

Projekte

Es gibt unzählige Anwendungen für Maps und Sets. Um ein bisschen ein Gefühl zu entwickeln wofür sie eigentlich gut sein könnten sehen wir uns ein paar einfache Beispiele an.



StateLookup

Eine typische Anwendung für HashMaps ist die Suche. Z.B. haben wir den Kürzel eines US-Bundesstaates (z.B. "NY") und würden gerne wissen um welchen Bundesstaat es sich denn handelt. Im Internet findet man Tabellen mit der Liste aller Bundesstaaten, z.B. in der Form:

```
AL,Alabama
AK,Alaska
AZ,Arizona

...
```

Wir lesen also Zeile für Zeile, und fügen diese in unsere HashMap ein:

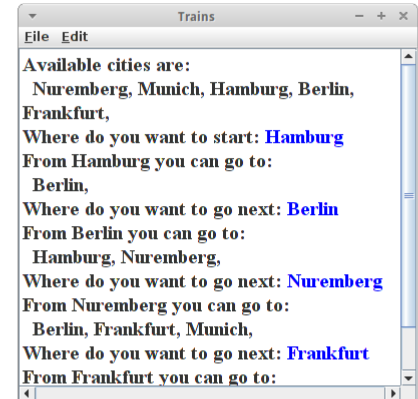
```
private void readStateEntry(String line) {
    int comma = line.indexOf(",");
    String stateInitial = line.substring(0, comma).trim();
    String stateName = line.substring(comma + 1).trim();
    states.put(stateInitial, stateName);

}
```

(das kann man natürlich auch mit der split() Methode oder dem StringTokenizer machen). Der Rest funktioniert dann genauso wie im Projekt Dictionary.

Dieses Projekt ist ein Beispiel für eine ganze Klasse von ähnlichen Lookups:

- Postleitzahl und Stadt
- Länder und der Hauptstädte
- Ländercodes und Länder

Trains



Eine typische Anwendung für HashMaps sind Fahrpläne. Nehmen wir an wir wollen von München nach Berlin und es gibt aber keine direkte Verbindung. Dann sehen wir im Fahrplan nach und sehen, dass man von München nach Nürnberg fahren kann, und von Nürnberg gibt es einen Zug nach Berlin. So eine einfacher Fahrplan könnte eine Textdatei sein, die alle Verbindung enthält:

```
Nuremberg > Berlin
Nuremberg > Frankfurt
Nuremberg > Munich
Munich > Nuremberg
Hamburg > Berlin
```

Wichtig ist hier, dass die Verbindungen zwischen Städten eine Richtung haben, also der Zug geht von Nürnberg nach Berlin, muss aber nicht zurück gehen.

Der nächste Schritt ist sich zu überlegen wie man so einen Fahrplan in einer HashMap unterbringt. Es ist klar, dass der Key der Ausgangsbahnhof sein muss. Da es aber mehrere Zielbahnhöfe geben kann, müssen wir hier eine Liste verwenden:

```
private HashMap<String, ArrayList<String>> connections;
```

Außerdem macht es auch noch Sinn eine Liste von allen Bahnhöfen irgendwo zu haben:

```
private ArrayList<String> cities;
```

Im *setup()* lesen wir also den Fahrplan und befüllen unsere beiden Datenstrukturen. Mit dem StringTokenizer trennen wir source von destination:

```
StringTokenizer st = new StringTokenizer(line, ">");
String source = st.nextToken().trim();
String destination = st.nextToken().trim();
```

Dann sollten wir checken, ob es den Ausgangsbahnhof schon gibt

```
if (!cities.contains(source)) {
    cities.add(source);
    connections.put(source, new ArrayList<String>());
}
```

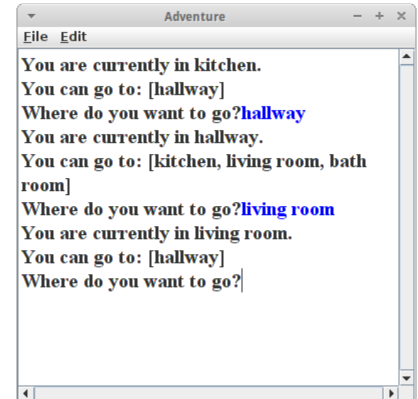
und schließlich müssen wir die neue Verbindung hinzufügen:

```
ArrayList<String> cits = connections.get(source);
cits.add(destination);
```

Nachdem die Daten jetzt geladen sind, können wir mit dem eigentlichen Programm fortfahren. Als erstes sollten wir dem Nutzer eine Liste aller Ausgangsbahnhöfe auflisten. Daraus sollte er seinen Ausgangbahnhof wählen. Im nächsten Schritt listen wir die möglichen Zielbahnhöfe auf, und lassen den Nutzer wieder wählen. Das machen wir so lange, bis der Nutzer seinen Zielbahnhof erreicht hat, also den leeren String eingibt.

Erweiterungen: Was natürlich cool wäre, wenn der Nutzer einfach nur seinen Ausgangsbahnhof und Zielbahnhof eingeben könnte, und das Programm dann automatisch eine Route vorschlägt. Im Kapitel zu Graphen lernen wir wie.

Adventure



Auch eine schöne Anwendung für HashMaps sind Abenteuer Spiele. In vielen von diesen text-basierten Spielen geht es darum eine Welt zu erkunden, und Gegenstände einzusammeln. Wir konzentrieren uns hier auf den Erkunden-Teil, aber der Einsammel-Teil ist auch nicht so schwer.

Ähnlich wie beim *Trains* Projekt benötigen wir eine Beschreibung der Umgebung. Am einfachsten ist da eine Beschreibung unserer Wohnung. Also bei uns zu hause sieht das so aus:

```
hallway > kitchen
hallway > living room
hallway > bath room
kitchen > hallway
living room > hallway
bath room > hallway

bath room > kitchen
```

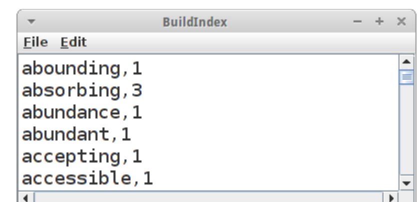
Auch hier verwenden wir wieder eine HashMap die den Plan unserer Wohnung wiedergibt:

```
private HashMap<String, ArrayList<String>> roomMap;
```

Da es sich um ein Erkundungsspiel handelt, benötigen wir keine Liste aller Räume, anstelle lassen wir den Spieler einfach in der Küche mit seiner Erkundung beginnen. Wir listen dann die Räume auf die von der Küche aus zu erreichen sind, und bitten den Spieler eine Wahl zu treffen. Auf diese Art und Weise kann der Spieler nach und nach unsere ganz Wohnung erkunden. Mit der Eingabe des leeren Strings endet das Spiel.

Erweiterungen: Für diesen Spieletyp gibt es zahllose Erweiterungen. Man könnte zum Beispiel die Welt aus StarWars oder Herr der Ringe auf diese Art abbilden. In den verschiedenen Räumen könnte man magische Gegenstände verstecken. Und manche Räume kann man nur betreten wenn man einen bestimmten Gegenstand hat, usw...

BuildIndex



Bücher aus Papier kann man nicht so leicht durchsuchen wie elektronische Bücher. Deswegen haben die meisten Bücher hinten einen Index, auch Stichwortverzeichnis genannt. Als Beispiel wollen wir eine Liste von Stichwörtern für das Buch "TomSawyer.txt" [4] erstellen.

Wie üblich gehen wir Zeile für Zeile durch das Buch und benutzen den StringTokenizer

```
StringTokenizer st = new StringTokenizer(line, "[]\"',;:.!?()-/
\t\n\r\f");
```

um die Wörter aus einer Zeile zu extrahieren. Das ist eines der wenigen Male wo die *split()* Methode der String Klasse nicht funktionieren würde (es sei denn man beherrscht Reguläre Ausdrücke).

Wir gehen also alle Zeilen und alle Wörter (Tokens) durch und speichern diese in einer HashMap

```
private Map<String, Integer> words = new HashMap<String, Integer>();
```

Diese HashMap befüllen wir dann mit der folgenden Methode:

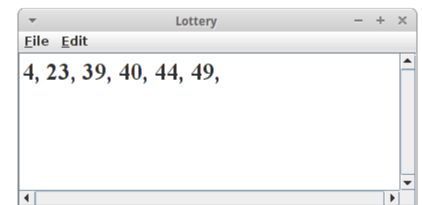
```
private void addWordToHashMap(String word) {
    if (word != null) {
        if (words.containsKey(word)) {
            int count = words.get(word);
            words.put(word, ++count);
        } else {
            words.put(word, 1);
        }
```

```
        }
    }
```

denn wir wollen zählen, wie häufig ein bestimmtes Wort vorkommt. Jetzt müssen wir nur noch die Map auf der Konsole ausgeben.

Für unseren BuildIndex gibt es ganz viel Verbesserungspotential:

- Wenn wir unsere Liste von Wörtern betrachten, stellen wir fest, dass die meisten Wörter mit weniger als acht Buchstaben eigentlich nichts in einem Index verloren haben. Also sollten wir sie gar nicht erst in die Liste mit aufnehmen.
- Man könnte noch Wörter die im Plural enden herausfiltern (das ist im Englischen relativ einfach).
- Man könnte Wörter mit nutzlosen Endungen (im Englischen "ly", "ial", "ive", "ous", "ed") herausfiltern.
- Man kann auch eine Liste von Stoppwörtern haben und diese dann herausfiltern [1].
- Sortieren: wenn wir anstelle von HashMap eine TreeMap verwenden, dann ist der Index auf einmal sortiert.
- Man könnte sich auch noch die Zeile (oder Seite) merken in der das Wort vorgekommen ist.

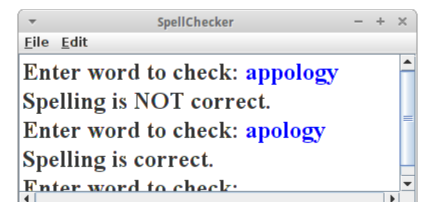


Lottery

In der Lottery geht es darum 6 Zufallszahlen zwischen 1 und 49 zu generieren. Selbstverständlich sollte es keine Duplikate geben. Am Einfachsten setzt man das mit einem Set um. Wenn man möchte, dass die Zahlen sortiert sein sollen, verwendet man ein *TreeSet*, ansonsten tut es auch ein normales *HashSet*.

```
private Set<Integer> generateLotteryNumbers() {
    Set<Integer> nrs = new TreeSet<Integer>();

    while ( nrs.size() < 6 ) {
        int r = rgen.nextInt(1, 49);
        nrs.add(r);
    }

    return nrs;
}
```

Man sollte sich wirklich überzeugen wie einfach diese Lösung ist indem man versucht das gleiche Problem mit einem Array oder einer Liste zu lösen.



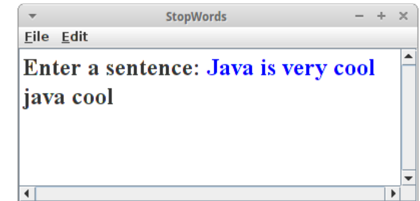
SpellChecker

Eine ganz einfache Anwendung für die *Set* Klasse ist ein SpellChecker. Dazu befüllen wir zunächst ein Set mit allen Wörter der englischen Sprache,

```
Set<String> words =
buildIndexFromFile("dictionary_en_de.txt");
```

und verwenden dann die *contains()* Methode um festzustellen ob das Wort richtig geschrieben wurde:

```
String word = readLine("Enter word to check: ");
if (words.contains(word.toLowerCase())) {
    println("Spelling is correct.");
} else {
    println("Spelling is NOT correct.");
}
```



StopWords

Wenn wir auf Google etwas suchen, dann nimmt Google nicht alle Wörter die wir tippen für die Suche, sondern entfernt erst einmal die sogenannten "Stop-Words" [1,2]: das sind Wörter die eigentlich keine Bedeutung für die Suche haben, wie z.B.

```
a about above after again against all am ...
```

Wir wollen also aus einem gegebenen Satz oder Suchanfrage die Stop-Words herausfiltern. Dazu verwenden wir am besten ein Set,

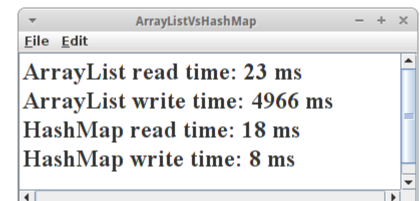
```
Set<String> stopWords = buildIndexFromFile("StopWords.txt");
```

das wir mit den Stoppwörtern befüllen. Danach gehen wir einfach Wort für Wort durch und checken ob es ein Stoppwort ist:

```
String sentence = readLine("Enter a sentence: ");
StringTokenizer st = new StringTokenizer(sentence);
while (st.hasMoreTokens()) {
    String word = st.nextToken().toLowerCase();
    if (!stopWords.contains(word)) {
        print(word + " ");
    }

}
```

Man kann genau das gleich Verfahren verwenden um Schimpfwörtern und/oder Obszönitäten zu finden und zu filtern.

Challenges



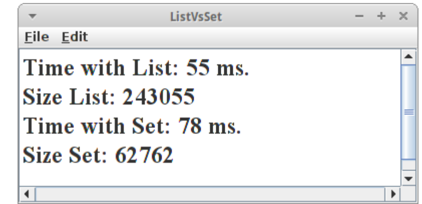
ArrayListVsHashMap

Wir haben schon mehrmals erwähnt, dass die HashMap sehr schnell ist, speziell im Vergleich zu Listen. Das müssen wir natürlich beweisen. Der Test ist identisch mit dem aus dem letzten Kapitel in welchem wir ArrayList und LinkedList verglichen haben. Der Test der HashMap ist beim Schreiben ein klein wenig anders, wir brauchen ja einen Key, dafür verwenden wir einfach den Zähler:

```
private static long testReadHashMap() {
    HashMap<Integer,Integer> al = new HashMap<Integer,Integer>();

    // fill list with some dummy data:
    for (int i = 0; i < 1000000; i++) {
        al.put(i, 42);
    }

    // start the read test:
    long start = System.currentTimeMillis();
    for (int i = 0; i < 100000; i++) {
        // read an element at a random position:
        int randomPos = (int) (100000.0 * Math.random());
        al.get(randomPos);
    }
    long end = System.currentTimeMillis();
    return (end - start);
```

```
    }
```

Beim Lesen schlägt sich die ArrayList noch recht wacker, aber beim Schreiben ist sie tausend mal langsamer. Das sollte genügen.



ListVsSet

Kommen wir zu unserem nächsten Duell: List vs Set. Es geht darum, dass wir feststellen wollen ob ein bestimmtes Wort in einem Text vorkommt. Als Text nehmen wir *Ulysses* von James Joyce [3]. In einem ersten Schritt lesen wir das Buch von Datei,

```
String text = readTextFromFile("Ulysses.txt");
```

und in einem zweiten Schritt bauen wir daraus einen Index, einmal mithilfe einer *ArrayList* und einmal mit einem *HashSet*:

```
List<String> indexList = buildIndexUsingList(text);

Set<String> indexSet = buildIndexUsingSet(text);
```

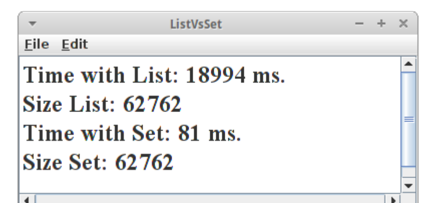
Wir verwenden den StringTokenizer,

```
private List<String> buildIndexUsingList(String text) {
    ArrayList<String> al = new ArrayList<String>();
    //HashSet<String> al = new HashSet<String>();
    StringTokenizer st = new StringTokenizer(text);
    long startTime = System.currentTimeMillis();
    while (st.hasMoreTokens()) {
        String word = st.nextToken();
        al.add(word);
    }
    long time = System.currentTimeMillis() - startTime;
    println("Time with List: " + time + " ms.");
    return al;

}
```

und fügen die Tokens der Liste oder dem Set hinzu. Der Code für das Set sieht genauso aus, nur ArrayList wird durch HashSet ersetzt.

Als erstes Resultat sehen wir, dass beim reinen Hinzufügen List und Set faktisch gleich schnell sind. Aber wir sehen auch, dass die Liste viel größer geworden ist als das Set: das hat damit zu tun, dass es in der Liste Duplikate gibt, die gibt es im Set nicht.

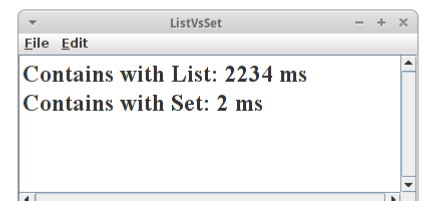
Da das eigentlich Platzverschwendung ist, fügen wir die Zeilen




```
if (!al.contains(word)) {
    al.add(word);

}
```

in den Code mit der ArrayList ein (ist beim Set ja nicht nötig). Jetzt sehen wir, dass beide gleich groß sind, aber jetzt braucht die List viel länger. Das wollen wir jetzt aber mal übersehen.

Kommen wir zu unserem eigentlichen Auftrag: wir sollen feststellen ob ein bestimmtes Wort in einem Text vorkommt oder nicht. Dafür gibt es die *contains()* Methode, sowohl bei der Liste als auch beim Set. Wir lassen das zehntausend mal durchlaufen und messen die Zeit:



```
private long testContainsList(List<String> al) {
    long start = System.currentTimeMillis();
    for (int i = 0; i < 10000; i++) {
        if (al.contains("marvel")) {
            // we found it!
        }
    }
    long end = System.currentTimeMillis();
    return (end - start);
}
```

Das Ergebnis spricht für sich: Das Set ist ca. tausendmal schneller!

HashList

Im Prinzip bräuchte man eigentlich gar keine ArrayLists, man könnte sie durch eine HashMap ersetzen. Um das zu demonstrieren implementieren wir eine Klasse HashList,

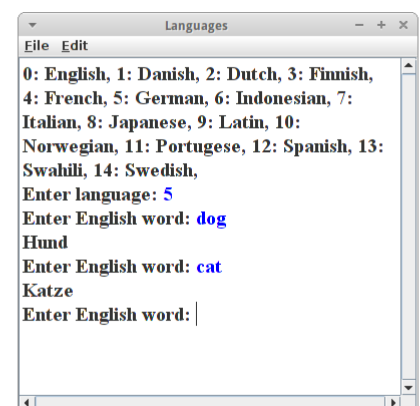
```
public class HashList {

    private HashMap<Integer, Object> map;
    private int position = -1;

    ...
```

die eine HashMap für die Datenhaltung verwendet, aber ansonsten ein List Interface implementiert, ähnlich wie unsere HomemadeArrayList Klasse im letzten Kapitel, d.h. folgende Methoden müssen wir implementieren:

- size()
- add(object)
- get(i)
- set(i, object)

Die *remove()* Methode lassen wir absichtlich wieder weg, da die etwas komplizierter ist.

Languages



Was ist schon ein Wörterbuch, das von einer Sprache in eine andere übersetzt? Wir wollen ein Wörterbuch, das von einer Sprache in zehn andere übersetzt! Dazu muss man aber erst mal irgendwo die nötigen Daten finden. Glücklicherweise gibt es auf dem Website zu dem Buch "Introduction to Programming in Java" von Robert Sedgewick und Kevin Wayne [5] eine Datei die die Übersetzungen von über 800 englischen Wörtern in zehn andere Sprachen enthält [6].

Die Datei "Languages.csv" enthält diese Daten:

```
"cat","kat","kattekop","kissa","chat, matou,
rosse","Katze",...
```

Das erste Wort in jeder Zeile ist das englische Wort, gefolgt von der dänischen, der holländischen, usw., Übersetzung. Welche Sprache an welcher Stelle kommt steht in der ersten Zeile der Datei. Die Daten zu parsen wird nicht ganz einfach, wenn wir uns die französische Übersetzung für Katze ansehen, denn es gibt anscheinend mindestens drei Worte für Katze. Aber wenn wir nach Anführungsstrichen mit der *indexOf()* Methode suchen,

```
private ArrayList<String> parseLine(String line) {
    ArrayList<String> translations = new ArrayList<String>();
    while (true) {
        int begin = line.indexOf("\"");
        if (begin < 0)
            break;
        int end = line.indexOf("\"", begin + 1);
        String s = line.substring(begin + 1, end);
        line = line.substring(end + 1);
        translations.add(s);
    }
    return translations;

}
```

dann ist das durchaus machbar. Die Methode *parseLine()* zerlegt also eine Zeile aus unserer Datei, und wandelt sie in eine ArrayList von Strings um. Diese ArrayList enthält also das englische Wort mit all seinen Übersetzungen. Deutsch ist an sechster Stelle, d.h. mit

```
ArrayList<String> translations = parseLine(line);

String german = translations.get(5);
```

erhalten wir die deutsche Übersetzung des Wortes. So Parsen ist erledigt.

Ähnlich wie bei unserem einfach Dictionary Projekt, wollen wir ja nach Wörtern suchen, und dafür verwenden wir die HashMap:

```
private Map<String, List<String>> dictionary;
```

Allerdings speichern wir jetzt eben nicht nur ein Wort pro englischem, sondern zehn, und deswegen steht da auch *List<String>*. Die Liste zu befüllen ist ganz einfach:

```
ArrayList<String> translations = parseLine(line);

dictionary.put(translations.get(0), translations);
```

Um zu übersetzen müssen wir nur wissen welche Sprache gewünscht ist (also z.B. 5 für Deutsch) und welches Wort übersetzt werden soll:

```
private String translate(String english, int lang) {
    List<String> words = dictionary.get(english);
    if (words != null) {
        return words.get(lang);
    }
    return null;

}
```

Das *if* ist nötig um zu verhindern, dass unser Programm abstürzt falls wir nach einem Wort suchen, das nicht in unserer Datenbank ist.

Mögliche Erweiterungen zu unserem "Uber"setzungsprogamm könnten sein:

- Wie könnte man von jeder der zehn Sprachen in jede andere der zehn Sprachen übersetzen?
- Könnte man nicht nur Wörter, sondern ganze Sätze übersetzen?
- Man könnte natürlich auch eine hübsche UI Anwendung dafür schreiben.

Bag

Manchmal benötigt man eine Datenstruktur die es bei Java gar nicht gibt. Ein Beispiel ist die *Bag*, auch Multiset, oder auf Deutsch "Multimenge" genannt. Dabei handelt es sich im Prinzip um ein Set das seine Elemente zählt, oder anders ausgedrückt, ein Set das auch Duplikate erlaubt. Eine praktische Anwendung hätten wir schon gehabt, und zwar das Histogram im letzten Kapitel oder BuildIndex in diesem.

```
Console ⊠  Problems  @ Javadoc  Dec
<terminated> HashBag (1) [Java Application] /usr/l
6
0
-1
Exception in thread "main" java.lang.
        at containers.HashBag.remove(
        at containers.HashBag.main(Ha
```

Da es die *Bag* in Java nicht gibt, müssen wir die halt selbst schreiben. Überlegen wir uns wie das Interface der Bag aussehen sollte, also was sie alles können soll:

- **add(object):** füge ein neues *object* Element hinzu;
- **add(object, n):** füge *n* neue *object* Elemente hinzu;
- **remove(object, i):** entferne *n* Elemente;
- **getCount(object):** wieviele *object* Elemente sind in der Bag.

Ganz richtig ist das nicht, denn eigentlich ist eine Bag ja ein Set, und Sets sollen ja *addAll()*, *retainAll()*, *removeAll()* und *containsAll()* können. Also müsste unsere Bag das auch. Das wird aber jetzt ein bisschen viel, deswegen bleiben wir beim Wesentlichen.

Mit einem kleinen Beispiel Code wird sofort klar wie eine Bag funktioniert:

```
HashBag<String> b = new HashBag<String>();
b.add("a", 8);      // add eight times "a"
b.remove("a", 2);   // remove two of the "a"
System.out.println(b.getCount("a"));
b.remove("a", 7);   // remove seven of the "a" (we only have 6 left!)
System.out.println(b.getCount("a"));

System.out.println(b.getCount("b"));
```

Es gibt jetzt zwei Möglichkeiten die Bag umzusetzen, entweder via Composition oder via Generalization.

Composition

Composition bedeutet, dass wir mit einer Instanzvariable arbeiten. Mit etwas Glück finden wir eine Klasse die schon das meiste von dem was wir brauchen kann. Und wir haben Glück, denn die HashMap eignet sich hervorragend um als Bag verwendet zu werden. Wir deklarieren also unsere Klasse und instanziieren die Map,

```
public class HashBag<K> {

    private HashMap<K, Integer> map;

    public HashBag() {
        map = new HashMap<K, Integer>();
    }

    ...
```

In die Bag wollen wir Instanzen vom Typ *K* speichern, wobei *K* für irgendeine Klasse steht, z.B. String. Unsere Map hat Integer als Wert, das ist praktisch für's Zählen. Implementieren wir die *add()* Methode:

```
    public void add(K key, int i) {
        if (map.containsKey(key)) {
            i += map.get(key);
        }
        map.put(key, i);

    }
```

Wir stellen erst fest, ob wir vielleicht schon gleiche Elemente in unserer Liste haben, falls ja müssen wir deren Anzahl dazu addieren. Ansonsten speichern wir einfach den Schlüssel mit der Anzahl in unserer Map. Die *getCount()* ist auch ganz einfach:

```
public int getCount(K key) {
    if (map.containsKey(key)) {
        return map.get(key);
    }
    return -1;
}
```

Falls wir ein Element haben, geben wir einfach den Wert zurück den wir gespeichert haben, ansonsten -1, wenn es den Key gar nicht gibt. Bleibt noch *remove()*:

```
public void remove(K key, int i) throws Exception {
    if (map.containsKey(key)) {
        int count = map.get(key) - i;
        if (count < 0) {
            count = 0;
        }
        map.put(key, count);
    } else {
        throw new Exception("bag does not contain element " + key);
    }
}
```

Hier müssen wir zwei Dinge beachten: darf die Zahl eines Elements negativ werden: Wir sagen mal Nein. Und was passiert, wenn es das Element gar nicht gibt? Wir übertreiben hier ein bisschen und schmeißen eine Exception. Das muss man aber nicht, man kann das auch einfach ignorieren. Das war's.

Generalization

Generalization bedeutet, dass wir Vererbung verwenden. Wir sagen also:

```
public class HashBag2<K> extends HashMap<K, Integer> {

    public HashBag2() {
        super();
    }

    ...
```

also unsere HashBag2 ist eine HashMap. Implementieren wir die *add()* Methode:

```
public void add(K key, int i) {
    if (this.containsKey(key)) {
        i += this.get(key);
    }
    this.put(key, i);

}
```

die sieht genauso aus wie oben, nur *map* wurde durch *this* ersetzt. Das Gleiche ist bei der *getCount()* Methode:

```
public int getCount(K key) {
    if (this.containsKey(key)) {
        return this.get(key);
    }
    return -1;

}
```

und auch bei der *remove()* Methode:

```
public void remove(K key, int i) throws Exception {
    if (this.containsKey(key)) {
        int count = this.get(key) - i;
        if (count < 0) {
            count = 0;
        }
        this.put(key, count);
    } else {
        throw new Exception("bag does not contain element " + key);
    }

}
```

Was ist jetzt der Unterschied zwischen Composition und Generalization? Im ersten Fall kann die *HashBag* nur das was wir ihr beigebracht haben, also es gibt nur die *add()*, *getCount()* und *remove()* Methoden. Beim zweiten Fall ist das anders: die *HashBag2* kann viel mehr, und zwar alles was die *HashMap* kann. Das kann praktisch sein, z.B. wenn wir durch alle Element iterieren wollen:

```
for (String word : b.keySet()) {
    System.out.println(word+": "+b.getCount(word));

}
```

Das geht mit der *HashBag* nicht. Dort müssten wir das alles noch implementieren. Der Nachteil der Vererbung liegt aber darin, dass alle Methoden der *HashMap* zur Verfügung stehen, also z.B. auch *containsValue()* etc. sind verwendbar was evtl. nicht gewünscht ist.

Abschließend sei noch bemerkt, dass es ausser den Collections die Java zur Verfügung stellt, auch noch andere Bibliotheken gibt die zusätzliche Collections zur Verfügung stellen, z.B. die *Apache Commons* Bibliothek [7] oder auch der *Guava* Teil der Google Core Libraries for Java [8].

MultiMap

In einer HashMap kann ein Schlüssel immer nur einen Wert haben. In der *MultiMap* kann es zu einem Key auch mehrere Werte geben. Als Beispiel betrachten wir ein Wörterbuch, in dem es zu einem Wort mehrere Übersetzungen geben kann:

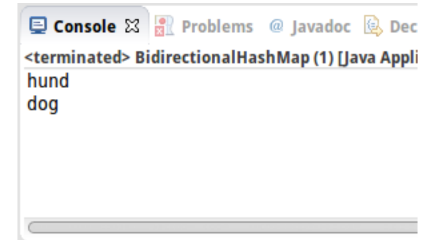
```
Console ⊠  Problems  @ Javadoc  Dec
<terminated> MultiHashMap (1) [Java Application]
[Reise, Trip, Fahrt, Ausflug]
[Reise, Trip, Fahrt]
```

```
MultiHashMap<String, String> map = new
MultiHashMap<String, String>();
map.put("trip", "Reise");
map.put("trip", "Trip");
map.put("trip", "Fahrt");
map.put("trip", "Ausflug");
Collection c = map.get("trip");
System.out.println(c);
map.remove("trip", "Ausflug");   // just remove one of the values
c = map.get("trip");

System.out.println(c);
```

Das Einfachste ist es Composition zu verwenden. Wir müssen lediglich überlegen welche Datenstruktur unseren Anforderungen am nähesten kommt. Eine Möglichkeit ist eine Map mit Listen als Werte zu nehmen:

```
public class MultiHashMap<K, V> {

    private HashMap<K, List<V>> map;

    public MultiHashMap() {
        map = new HashMap<K, List<V>>();
    }
    ...
}
```

Der Rest ist dann ein Klacks. Aber natürlich kann man auch Generalization verwenden.



BidirectionalMap

Bei einer Map kann man immer nur ein eine Richtung suchen: ich kann nur nach dem Schlüssel suchen, nie nach dem Wert. Bei einer BidirectionalMap geht das aber: da kann man sowohl nach den Schlüsseln als auch nach den Werten suchen. Z.B. in einem Wörterbuch, möchten wir sowohl nach den englischen als auch nach den deutschen Wörtern suchen können:

```
BidirectionalHashMap<String, String> map = new
BidirectionalHashMap<String, String>();
map.put("dog", "hund");
map.put("cat", "katze");
map.put("fish", "fisch");
map.remove("fish");
System.out.println(map.get("dog"));

System.out.println(map.getKey("hund"));
```

Wir verwenden wieder Composition. Hier ist die Idee zwei Maps als Instanzvariablen zu verwenden, und zwar eine für die Vorwärtssuche und eine für die Rückwärtssuche:

```
public class BidirectionalHashMap<K, V>{

    private HashMap<K, V> forwardMap;
    private HashMap<V, K> reverseMap;

    public BidirectionalHashMap() {
        forwardMap = new HashMap<K, V>();
        reverseMap = new HashMap<V, K>();
    }
    ...
}
```

Wenn immer wir also einen Wert einfügen oder löschen, müssen wir das in beiden Maps machen. Der Rest ist wieder ganz einfach.

Research

In diesem Kapitel gibt es wieder ein bisschen was zu erforschen.

Animations

Im Internet finden sich zahlreiche Beispiel die versuchen verschieden Datenstrukturen und Algorithmen zu visualisieren. Versuchen Sie zwei oder drei Sites zu finden, die die eine oder andere Datenstruktur visualisieren.

Minecraft

Wie groß ist das Spielfeld von Minecraft? D.h. wie weit kann man denn in die x-, y- und z-Richtung gehen? Wenn wir das grob wissen, können wir abschätzen wieviel Speicher (RAM) ein Array benötigen würde in dem wir das gesamte Spielfeld speichern könnten. Selbst wenn wir annehmen, dass wir nur ein Byte pro Spielfeld Speicher benötigen, werden wir feststellen, dass unser Speicher nicht ausreichen würde. Wie machen die das dann? (Hinweis: Map)

SpellChecker

Wir haben oben ja einen einfachen SpellChecker geschrieben. Wie funktionieren denn die echten? Dazu können wir etwas in Referenz [9] nachlesen.

Fragen

1. Schreiben Sie Pseudo-Code für ein Wörterbuch.

2. Beschreiben Sie in Ihren eigenen Worten was die drei Datencontainer List, Map und Set voneinander unterscheidet.

3. BlackLists werden z.B. verwendet um Webseiten zu filtern, oder Kreditkarten die gestohlen wurden oder aus anderen Gründen ungültig sind zu erkennen. Welche Datenstruktur verwendet man am besten um eine BlackList umzusetzen?

4. Welche Datenstruktur würden Sie verwenden um Duplikate zu vermeiden?

5. Bei der Rechtschreibprüfung (spell checker) geht es darum festzustellen ob ein Wort richtig geschrieben wurde. Welche Datenstruktur ist dafür am besten geeignet? Gehen Sie davon aus, dass es sich um eine einfache Sprache wie das Englische handelt. (Mit kompliziert sind die Endungen der Wörter gemeint.)

6. Ihre Firma hat Sie beauftragt ein Programm zu schreiben, dass einen Text aus dem Englischen ins Deutsche übersetzen kann. Dabei geht es nicht um eine grammatikalisch richtige Übersetzung, sondern Sie sollen lediglich Wort für Wort übersetzen. Im Internet haben Sie eine Datei gefunden, die 50.000 englische-deutsche Wörterpaare enthält. Beschreiben Sie grob wie Sie das Problem lösen würden. Welche Datenstruktur würden Sie verwenden?

7. Sie sollen einen Generator für Lottozahlen (6 aus 49) schreiben. Welche Datenstruktur ist dafür am besten geeignet?

8. Beim Auswählen einer Datenstruktur, können Sie zwischen einem Array, einer Arraylist, einer LinkedList und einer HashMap wählen. Welche dieser vier Datenstrukturen sollten Sie eigentlich nie benutzen? Welche dieser vier Datenstrukturen ist in 80% aller Fälle die richtige Wahl?

9. Ihre Firma hat Sie beauftragt ein Programm zu schreiben, dass feststellen kann ob ein gegebener Vorname männlich oder weiblich ist. Im Internet haben Sie zwei Dateien gefunden, eine mit männlichen, die andere mit weiblichen Vornamen. Beschreiben Sie grob wie Sie das Problem lösen würden. Welche Datenstruktur würden Sie verwenden?

Referenzen

Die folgenden Referenzen sind relevant für dieses Kapitel.

[1] Stop words, https://en.wikipedia.org/wiki/Stop_words

[2] Google stop word list, https://meta.wikimedia.org/wiki/Stop_word_list/google_stop_word_list

[3] Ulysses, James Joyce, https://en.wikipedia.org/wiki/Ulysses_(novel)

[4] Tom Sawyer, https://en.wikipedia.org/wiki/Tom_Sawyer

[5] Introduction to Programming in Java, von Robert Sedgewick und Kevin Wayne

[6] Real-World Data Sets, introcs.cs.princeton.edu/java/data/

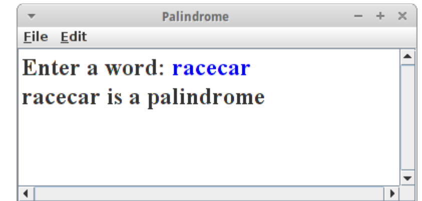
[7] Apache Commons Collections, https://commons.apache.org/proper/commons-collections/

[8] Guava: Google Core Libraries for Java, https://github.com/google/guava

[9] Kukich, Karen (1992). "Techniques for Automatically Correcting Words in Text" (PDF). ACM Computing Surveys 24 (4).

Recursion

Das Wort *Rekusion* kommt von dem lateinischen "recurrere" was "zurückkehren" bedeutet. Für uns heißt es soviel wie laufen wir mal los bis wir auf etwas stoßen, das wir schon kennen, und dann *kehren wir zurück* zu dem was wir vorher machen sollten. Rekursion ist eine weit verbreitete Technik zur Lösung aller möglichen Probleme. Im Vergleich zur *Iteration*, ist sie zwar häufig etwas langsamer, aber in der Regel viel eleganter. Fast alle Probleme lassen sich sowohl durch Iteration als auch durch Rekusion lösen.



Palindrome

Ein sehr schönes Beispiel für Rekursion sind *Palindrome* [1]: also Worte oder Sätze die das Gleiche bedeuten, egal ob man sie von links nach rechts oder rechts nach links liest:

- rentner
- lagerregal
- racecar
- was it a car or a cat i saw
- Saippuakippokukkakivikakkukoppikauppias (Finisch für "Soap-bowl-flower-stone-cake-box seller") [2]

Man kann ein Palindrom folgendermaßen definieren:

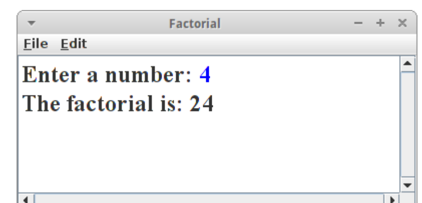
1. ein Wort das null oder einen Buchstaben lang ist, ist immer ein Palindrom;
2. ein Wort ist dann ein Palindrom, wenn der erste und letzte Buchstabe gleich sind, und wenn das Wort ohne die beiden auch ein Palindrom ist.

Das ist das typische Muster für Rekursion: wir haben immer einen *recursive case*, hier Schritt 2, in dem wir das Problem durch eine einfachere Version von sich selbst ausdrücken, und einen *base case*, hier Schritt 1, der dafür sorgt, dass die Rekursion irgendwann aufhört, deswegen nennt man es auch die Abbruchbedingung. Denn die häufigste Bug bei rekursiven Programmen ist, dass sie nie aufhören, so wie beim Infinite Loop aus dem letzten Semester.

Natürlich wollen wir das gleich in Java umsetzen: wir wollen eine Methode namens *isPalindrome(String s)* schreiben, die feststellt ob ein gegebener String ein Palindrom ist:

```
private boolean isPalindrome(String s) {
    if (s.length() <= 1) {    // base case
        return true;
    } else {                  // recursive case
        if (s.charAt(0) == s.charAt(s.length() - 1)) {
            return isPalindrome(s.substring(1, s.length() - 1));
        } else {
            return false;
        }
    }

}
```

Übrigens gibt es Palindrome auch in der Musik, und interessanterweise auch in unserer DNA: anscheinend speichert unserer Immunsystem die RNA von Viren sowohl vorwärts als auch rückwärts um böse Viren zu erkennen.



Factorial

Kommen wir zu einem anderen Klassiker der sich sehr schön mittels Rekursion lösen lässt, die Fakultät einer Zahl. In der Schule haben wir gelernt, dass man die Fakultät von vier wie folgt berechnet:

```
4! = 4 * 3 * 2 * 1
```

Wenn wir das als Programm schreiben wollen, dann bietet sich eine Schleife an:

```
int fac = 1;
for (int i=1; i<=4) {
     fac = fac * i;

}
```

Das ist die iterative Art und Weise die Fakultät einer Zahl zu berechnen. Es gibt aber noch eine andere, die rekursive. Dazu beobachten wir, dass:

```
4! = 4 * 3 * 2 * 1 = 4 * 3!
```

Wir können also 4! durch vier mal 3! ausdrücken. Im Allgemeinen gilt sogar

```
n! = n * (n-1)!
```

Immer wenn wir eine derartige Beziehung finden, also dass eine Funktion in *f(n)* durch eine Funktion in *f(n-1)* ausgedrückt werden kann, dann haben wir eine rekursive Lösung für unser Problem.

Setzen wir das gleich mal in Code um:

```
int factorial(int n) {
    return n * factorial( n-1 );
}
```

Wir haben also eine Methode die sich selbst aufruft: das ist Rekursion.

Unser Code hat allerdings ein kleines Problem: er hört nicht auf zu laufen, weil wir die Abbruchbedingung, den *base case*, vergessen haben. Bei der Fakultät ist das die Tatsache, dass per definitionem

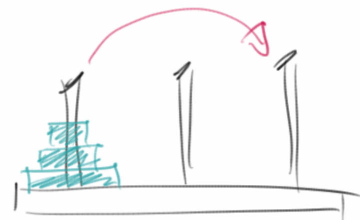
```
0! = 1
```

Damit sieht unser Java dann wie folgt aus:

```
int factorial(int n) {
    if ( n == 0 )
        return 1;
    else
        return n * factorial( n-1 );
}
```

Wir sehen also man kann die Fakultät einer Zahl sowohl durch Iteration als auch durch Rekursion berechnen.

Tower of Hanoi



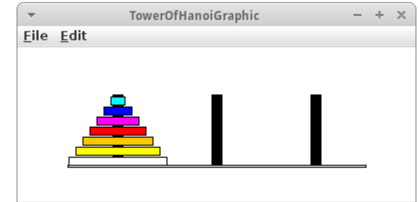
Das Rekursion nicht nur mit Wörtern oder Zahlen zu tun hat, sondern sehr häufig hübsche grafische Anwendungen hat, soll das Beispiel *Tower of Hanoi* zeigen [3]. Im Kindergarten hat eigentlich schon mal fast jeder mit dem Spiel *Türme von Hanoi* zu tun gehabt. Dabei geht es darum einen Stapel Scheiben, meist aus Holz, von dem Stab auf er linken Seite auf den Stab auf der rechten Seite zu verschieben. Dabei gilt es allerdings folgende Regeln zu beachten:

1. man kann immer nur eine Scheibe verschieben;
2. es kann immer nur die oberste Scheibe eines Stapels verschoben werden;
3. es darf nie ein größere Scheibe auf einer kleineren zu liegen kommen.

Man denkt, das kann doch gar nicht so schwer sein, aber beim ersten Mal ist es gar nicht so einfach.

Interessant für uns ist, dass es eine überraschend elegante rekursive Lösung für das Problem gibt: Angenommen, wir haben einen Stapel mit sieben Scheiben. Und nehmen wir an wir wüssten wie man einen Stapel mit sechs Scheiben verschieben kann. Dann ist die Lösung für unser Problem ganz einfach: Schiebe zunächst die sechs Scheiben auf den mittleren Stab. Dann nimm die übriggebliebene siebte Scheibe (die weiße) und verschiebe sie auf den rechten Stab. Und jetzt verschieben wir den Stapel mit den sechs Scheiben auf den rechten Stab. Das Problem mit sieben Scheiben ist gelöst. Wir können also das 7 Scheiben Problem lösen, wenn wir das 6 Scheiben Problem lösen können. Und das ist eine rekursive Lösung. Bleibt die Frage nach dem Abbruchkriterium: das ist ganz einfach: einen Stapel mit einer Scheibe können wir immer bewegen.

Also sieht unser Algorithmus wie folgt aus:




```
private void moveTower(int n, int source, int
destination, int temp) {
    if (n > 0) {
        moveTower(n - 1, source, temp,
destination);
        moveOneDisk(source, temp);
        moveTower(n - 1, destination, source, temp);
    }
}
```

Und das kann man auch sehr hübsch animieren.

Eine kleine Anmerkung zum *Tower of Hanoi* Problem: schon für relativ wenige Scheiben, dauert es sehr lange die Scheiben zu verschieben. Z.B., wenn man annimmt, das es eine Sekunde dauert eine Scheibe zu verschieben, dann dauert es 12 Tage um einen Stapel mit nur 20 Scheiben zu verschieben. Und für einen Stapel mi 60 Scheiben würde man mehr Zeit benötigen als unser Universum alt ist [3]!

Review

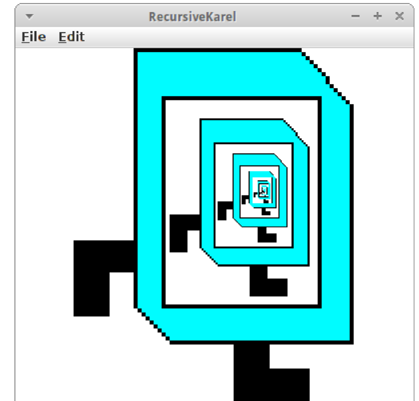
Mittels Rekursion lassen sich sehr viele Probleme elegant lösen. Sehr vielen Algorithmen denen wir begegnen werden bedienen sich daher eines rekursiven Ansatzes. Der folgt immer den folgenden zwei Schritten:

1. ein Problem kann durch eine einfachere Version von sich selbst ausgedrückt werden;
2. es gibt ein Abbruchkriterium, also irgend einen einfachen Fall von dem man die Antwort kennt.

Sehr, sehr viele Problem lassen sich durch Rekursion lösen.

Projekte

Man mag sich vielleicht die Frage stellen, muss ich mir die Rekursion wirklich antun? Nach den folgenden Beispielen kann jeder die Frage für sich selbst beantworten. Für mich ist die Antwort aber ziemlich eindeutig: Rekursion ist cool.



RecursiveKarel

In dem Projekt geht es darum Rekursion zu visualisieren. Dazu nehmen wir ein Bild von Karel, welches wir schrittweise immer kleiner machen, und die einzelnen Bilder ineinander zeichnen. Wir beginnen mit dem großen Karel:

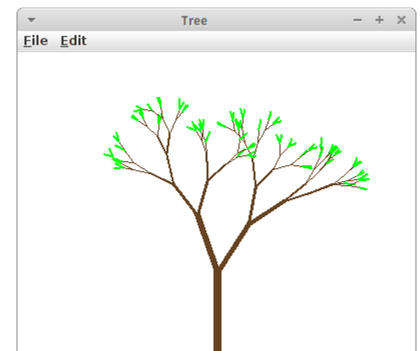
```
drawKarel(4.0, 10, -40);
```

In der drawKarel() Methode, zeichnen wir dann ein GImage, skaliert wie gewünscht,

```
private void drawKarel(double scale, int x, int y)
{
    GImage karel = new GImage("Karel0.png");
    karel.scale(scale);
    add(karel, (SIZE-karel.getWidth())/2 + x, (SIZE-karel.getHeight())/2
+ y);

    if (karel.getWidth() < 2) { // base case
        return;
    } else { // recursive case
        drawKarel(scale / 2, x+=(5*scale), y-=(2.5*scale));
    }

}
```

und rufen danach uns selbst wieder auf, allerdings soll das Bild dieses mal nur halb so groß sein, und etwas verschoben.

Hier handelt es sich um eine ganz einfaches Beispiel von Selbstähnlichkeit, viele Fraktale beruhen auf einem ganz ähnlichen Prinzip.



Tree

Bäume zu zeichnen ist eines von den Beispielen, die ganz einfach sind mit Rekursion, aber ziemlich knifflig werden wenn man es mit Iteration versucht. Ein Baum besteht aus Ästen, und wir zeichnen diese rekursiv, einen nach dem anderen. Wir schreiben wieder ein GraphicsProgram und beginnen damit den ersten Ast (also den Stamm) zu zeichnen:

```
drawBranch(x, y, angle, length);
```

dabei sind *x* und *y* die Position wo ein Ast beginnt, *angle* ist der Winkel mit dem sich der Ast neigt, und *length* ist die Länge des Astes. In jeder *drawBranch()* Methode zeichnen wir also zunächst einen Ast als GLine,

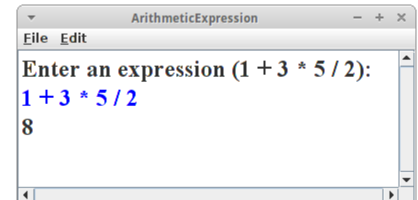
```
private void drawBranch(double x0, double y0, double angle,
        double length) {

    double x1 = x0 - Math.cos(angle) * length;
    double y1 = y0 - Math.sin(angle) * length;

    drawLine(x0, y0, x1, y1, length);
```

Danach wollen aber rekursive jeweils zwei neue Zweige zeichnen, die ein klein bisschen kürzer sein sollen, und sich auch zufällig ein bisschen mehr nach links und rechts neigen sollen:

```
    // base case
    if (length < 10)
        return;
```

```
            // recursive case
            double bendAngle = Math.toRadians(rgen.nextDouble(-10, 10));
            double branchAngle = Math.toRadians(rgen.nextDouble(-30, 30));
            drawBranch(x1, y1, angle + bendAngle - branchAngle,
                    length * rgen.nextDouble(0.6, 0.8));
            drawBranch(x1, y1, angle + bendAngle + branchAngle,
                    length * rgen.nextDouble(0.6, 0.8));

        }
```

Wir brauchen natürlich auch ein Abbruchkriterium, und das ist sobald die Äste kürzer als zehn Pixel sind hören wir auf.



ArithmeticExpression

Die Berechnung von arithmetischen Ausdrücken ist auch etwas was sich sehr schön mittels Rekursion lösen lässt. Betrachten wir dazu einen Ausdruck wie

```
1 + 3 * 5
```

als Beispiel. Zunächst einmal ist das ein String den wir vom Nutzer erhalten

```
String expression = readLine("Enter an expression:");

println( evaluate( expression ) );
```

und *evaluate()* ist die rekursive Methode die den eingegeben String auswerten soll.

Beginnen wir mit dem *base case*: der tritt dann ein wenn unser String keine Operatoren enthält, dann ist es einfach eine Zahl, und wir können die einfach in einen *int* umwandeln:

```
private int evaluate(String expression) {
    // base case
    if (!expression.contains("+") && !expression.contains("-") && !
expression.contains("*")
            && !expression.contains("/")) {
        return Integer.parseInt(expression.trim());
    }
    ...

}
```

Kommen wir zum *recursive case*: wir wissen zwar nicht wie wir "1 + 3 * 5" ausrechnen können, aber wenn wir wüssten was "1" ist und was "3*5" ist, dann könnten wir es. D.h., wir berechnen erst einmal diese beiden, und kehren dann zurück (recur) zur Addition dieser beiden Terme. Deswegen spliten wir erst einmal den String beim '+',

```
int i = expression.indexOf('+');
String o1 = expression.substring(0, i);

String o2 = expression.substring(i + 1, expression.length());
```

und berechnen dann rekursiv:

```
int result = evaluate(o1) + evaluate(o2);
```

d.h. der rechte Term beinhaltet *evaluate("1")* und der rechte *evaluate("3*5")*. Die werden jetzt beide wiederum rekursiv ausgewertet, bis wir jeweils beim *base case* ankommen. So motiviert betrachten wir den *recursive case*:

```
        // recursive case
        int i = findPlusAndMinus(expression);
        if (i < 0) {
            i = findTimesAndDivideBy(expression);
        }

        String o1 = expression.substring(0, i);
        String o2 = expression.substring(i + 1, expression.length());

        int result = 0;

        switch (expression.charAt(i)) {
        case '+':
            result = evaluate(o1) + evaluate(o2);
            break;
        case '-':
            result = evaluate(o1) - evaluate(o2);
            break;
        case '*':
            result = evaluate(o1) * evaluate(o2);
            break;
        case '/':
            result = evaluate(o1) / evaluate(o2);
            break;
        }
        return result;
    }
```

Wir suchen als erstes nach einem Plus oder einem Minus. Falls wir eines finden, dann schneiden wir den String auseinander, und je nachdem ob es ein Plus oder ein Minus ist, addieren oder sutrahieren wir das Ergebnis der *evaluate()* Methode mit den jeweilgen linken und rechten Teilstrings. Falls es kein Plus oder Minus gibt suchen wir nach einem Mal oder Geteilt-Durch, und machen dann das Gleiche.

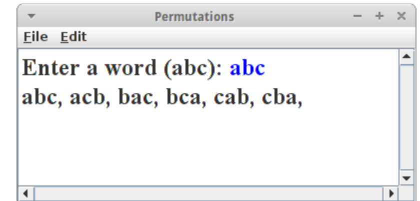
Obwohl es ja Punkt-vor-Strich heißt, und man naiv erwarten würde, dass man erst nach Mal und Geteilt-Durch sucht, und danach erst nach Plus und Minus, ist es in der Rekursion genau umgekehrt. Das hat damit zu tun, dass eben die rekursiven Schritte in umgekehrter Reihenfolge ausgeführt werden. Daran muss man sich erst gewöhnen, und das macht die Rekursion auch manchmal etwas gewöhnungsbedürftig.

Interessanterweise, passiert dieses Zurückkehren glech nochmal: wenn wir "3*5/2" betrachten, dann soll von links nach rechts ausgewertet werden, also erst "3*5" und dann "15/2" (ergibt 7 in Ganzzahl-Arithmetik). Das bedeutet aber, da wir ja rekursiv arbeiten, dass wir in der *findTimesAndDivideBy()* Methode nicht von links nach rechts nach '*' oder '/' suchen, sondern von rechts nach links:

```
    private int findTimesAndDivideBy(String expression) {
        int i;
        for (i = expression.length() - 1; i >= 0; i--) {
            if (expression.charAt(i) == '*' || expression.charAt(i) == '/') {
                break;
            }
        }
        return i;
    }
```

Das Gleiche gilt auch für die *findPlusAndMinus()* Methode. Am besten man probiert es mal aus und sieht, dass es funktioniert. Dann kann man ja mal die naive Version versuchen, und wird feststellen, dass da was Falsches rauskommt. Und dann geht man am besten einfach Schritt-für-Schritt das einfache Beispiel "1 + 3 * 5" durch um zu sehen, wie das mit der Rekusion funktioniert. Danach hat man dann die Rekursion wirklich verstanden!

Permutations



Die Berechnung von Permutationen [4] ist auch ein Problem das sehr elegant mit Rekursion gelöst werden kann. Als Beispiel betrachten wir die Buchstabenkombination "abc". Wir wollen alle möglichen Permutationen dieser Buchstabenkombination auflisten, als da sind:

```
abc, acb,   bac, bca,   cab, cba
```

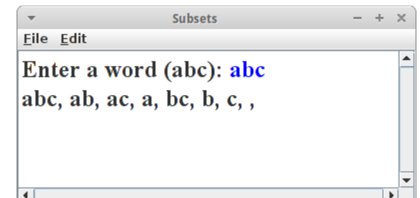
Wichtig bei Permutationen ist, dass es auf die Reihenfolge der Buchstaben ankommt.

Wir beginnen also indem wir den ersten Buchstaben festlegen, und permutieren dann die übrigen. Das sieht nach einem rekursiven Algorithmus aus: wir haben das größere Problem, permutiere eine Wort mit drei Buchstaben, auf ein einfacheres Problem, permutiere eine Wort mit zwei Buchstaben, zurückgeführt. Bleibt das Abbruchkriterium, und das sind Wörter die nur einen Buchstabe enthalten, da gibt es nur eine Permutation, der Buchstabe selbst.

```
private void permute(String picked, String remaining) {
    // base case
    if (remaining.length() == 1) {
        print(picked + remaining + ", ");
    }

    // recursive case
    for (int i = 0; i < remaining.length(); i++) {
        char pick = remaining.charAt(i); // pick a letter
        String front = remaining.substring(0, i);
        String back = remaining.substring(i + 1);
        permute(picked + pick, front + back);
    }

}
```

Was wir noch gerne wissen würden ist, wieviele Permutationen gibt es insgesamt?

Subsets



Bei Subsets [5] geht es um Untermengen: wir betrachten wieder die Buchstabenkombination "abc". Wir wollen alle möglichen Untermengen dieser Buchstabenkombination auflisten, auch das ist keine Raketenwissenschaft:

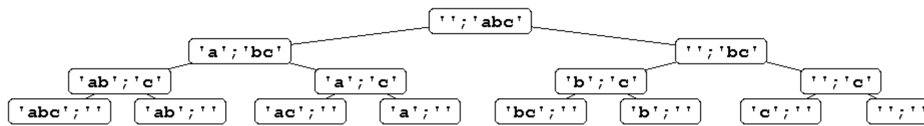
```
abc,   ab, ac, bc,   a, b, c
```

Bei Untermengen spielt die Reihenfolge der Buchstaben keine Rolle.

Die rekursive Lösung für dieses Problem ist nicht ganz offensichtlich, deswegen sehen wir uns erst mal den Code an, und versuchen ihn dann zu verstehen:

```
private void subset(String picked, String remaining) {
    // base case
    if (remaining.length() == 0) {
        print(picked + ", ");

    // recursive case
    } else {
        char pick = remaining.charAt(0); // pick first letter
        subset(picked + pick, remaining.substring(1));
        subset(picked, remaining.substring(1));
    }

}
```

Als Beispiel betrachten wir das Wort "abc": am Anfang enthält *picked* den Leerstring und *remaining* enthält "abc". Beim ersten Durchlauf, wird die Methode *subset()* zweimal aufgerufen, dabei einmal mit *picked="a"* und einmal mit *picked=""*, aber beide male mit *remaining="bc"*. Das kann man recht übersichtlich als Baum darstellen:

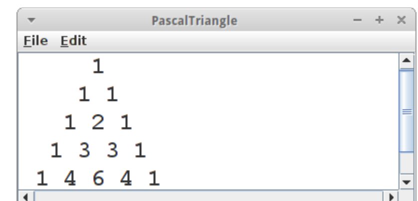


links steht immer der Wert von *picked*, und rechts der Wert von *remaining*. Wir sind fertig, wenn in *remaining* nichts mehr steht.

Unser Code liefert auch den Leerstring als mögliches Subset. Das ist eine überaus nicht-triviale Entscheidung, sie hat nämlich mit den Gödelschen Unvollständigkeitssätzen zu tun [23]: In einer Stadt (ist nicht Nürnberg), in der es nur einen Herrenfriseur gibt, gibt es eine Verordnung die besagt, dass jeder Mann der sich nicht selbst rasiert vom Friseur rasiert wird. Hat der Friseur einen Bart? Das ist die Frage die die Grundfesten der Mathematik erschüttert hat, und von der sich selbige bis heute nicht erholt hat.

Was wir noch gerne wissen würden ist, wieviele Subsets gibt es insgesamt?

PascalTriangle



Das Pascalsche Dreieck [6] ist auch ein sehr anschauliches Beispiel für Rekursion. Wenn wir irgendeine Zahl in dem Dreieck nehmen, z.B. die 4, dann ist sie die Summe der beiden Zahlen links und rechts in der Zeile darüber, also 1+3. Das gilt für alle Zahlen in dem Dreieck. Dass das eine rekursive Beziehung ist sehen wir daran, dass 3, wiederum die Summe der beiden Zahlen 1 und 2 ist, in der Zeile darüber. Wir nummerieren die Zeilen mit *n*, und die Spalten mit *k*, dann gilt also

```
private int pascalRecursion(int n, int k) {
    // recursive case
    return pascalRecursion(n - 1, k - 1) + pascalRecursion(n - 1, k);

}
```

Was ist das Abbruchkriterium? Wir sehen dass an den Seiten immer eine 1 steht. Für die linke Seite gilt immer *k==0* und für die rechte Seite gilt immer *k==n*. Also,

```
private int pascalRecursion(int n, int k) {
    if (k == 0 || k == n) { // base case
        return 1;
    } else { // recursive case
        return pascalRecursion(n - 1, k - 1) + pascalRecursion(n - 1, k);
    }

}
```

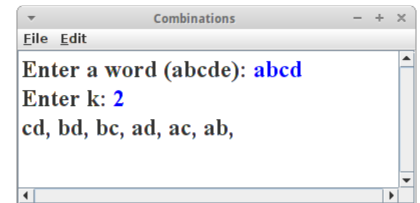
Damit lässt sich jetzt jede Zahl im Pascalsche Dreieck berechnen. Wenn wir noch über die Zeilen und Spalten iterieren, können wir das gesamt Dreieck darstellen.

Kommen wir zur Bedeutung des Pascalschen Dreiecks: es hat mit den Binomialkoeffizienten zu tun. Zur Erinnerung, was sind die Binomialkoeffizienten:

```
(a + b)¹ = 1*a  + 1*b
(a + b)² = 1*a² + 2*a*b  + 1*b²
(a + b)³ = 1*a³ + 3*a²*b + 3*a*b² + 1*b³
```

das sind die blauen Zahlen vor den Termen, und wenn wir diese mit den Zahlen im Pascalsche Dreieck vergleichen sollten wir erkennen, dass die sich sehr ähnlich sehen. Also stellt das Pascalsche Dreieck eine sehr einfache Art und Weise dar die Binomialkoeffizienten auszurechnen.

Interessant ist auch die Beziehung zwischen dem Pascalschen Dreieck und den Fibonacci-Zahlen [6].



Combinations

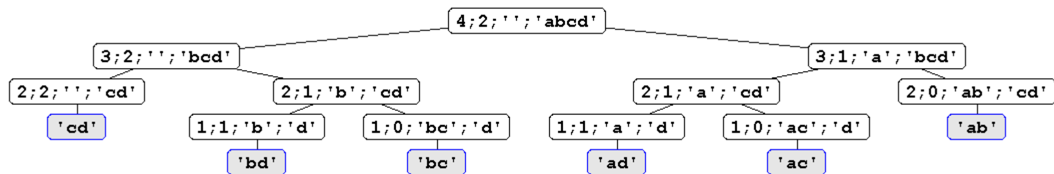
Das Pascalsche Dreieck ist nicht nur hübsch und hat alle möglichen interessanten mathematischen Eigenschaften, es hat auch eine durchaus praktische Anwendung, mit der wir nicht selten zu tun haben werden: manchmal sollen wir alle möglichen Paare aus einer Menge von z.B. vier Personen auflisten [7]. In dem Fall sagen wir, wir wollen alle möglichen Zweier-Kombination aus den vier Personen finden, man sagt auch 2 aus 4. Nehmen wir an die vier Personen heißen a, b, c und d, dann gibt es folgende Paarkombinationen:

```
ab, ac, ad, bc, bd, cd
```

Auch hier können wir wieder eine rekursive Beziehung erkennen:

```
private void combinations(int n, int k, String picked, String remaining)
{
    // base case
    if (k == 0) {
        print(picked + ", ");
    } else if ((k == n)) {
        print(picked + remaining + ", ");

        // recursive case
    } else {
        char pick = remaining.charAt(0); // pick first letter
        combinations(n - 1, k, picked, remaining.substring(1));
        combinations(n - 1, k - 1, picked + pick,
remaining.substring(1));
    }
}
```

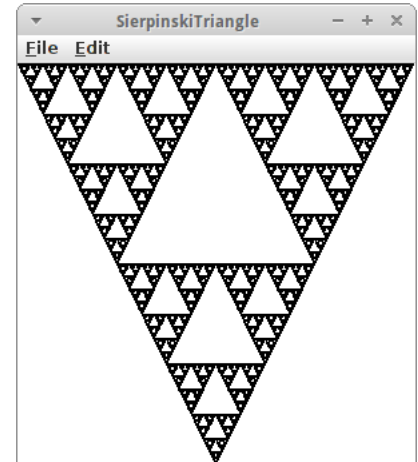
Der Code ist ähnlich zu dem von Subsets, allerdings durch die *n* und *k*, etwas komplizierter. Am besten wir betrachten wieder die Darstellung als Baum:



von links nach rechts stehen die Werte von *n*, *k*, *picked* und *remaining*. Nehmen wir an wir wollen alle Zweierkombination aus dem Wort "abcd". Dann ist *n=4* die Länge des Worts, *k=2* steht für Zweierkombinationen, und *picked* ist am Anfang leer, und *remaining* enthält "abcd".

Im ersten Schritt haben wir die Möglichket 'a' zu wählen (rechts) oder nicht zu wählen (links). Wenn wir 'a' gewählt haben, können wir als nächstes 'b' wählen (wieder rechts) oder 'b' nicht wählen (wieder links). Wenn wir 'b' gewählt haben, sind wir fertig, da wir ja schon zwei Buchstaben ('a' und 'b') gewählt haben. Wenn wir 'b' nicht gewählt haben, dann bleiben noch zwei Möglichkeiten: wir können 'c' wählen (rechts), oder 'c' nicht wählen (links). Wenn wir 'c' wählen, sind wir fertig weil wir 'a' und 'c' gewählt haben. Wenn wir 'c' nicht gewählt haben, sind wir auch fertig, weil es nur noch 'd' zu wählen gibt um einer Zweierkombination zu bilden, ansonsten wäre es nämlich eine Einserkombination.

Und was hat das jetzt mit dem Pascalsche Dreieck zu tun? Es sagt uns wieviele Kombinationen es gibt. Nehmen wir unser Beispiel mit 2 aus 4: 4 ist n, also die Zeile, und 2 ist k, also die Position in der Zeile. Wenn wir nachsehen, dann ist in der vierten Zeile an der zweiten Position die 6. (Wie üblich beginnen wir mit 0 zu zählen.) Und das ist auch was wir gefunden haben: 6 Zweierkombinationen.



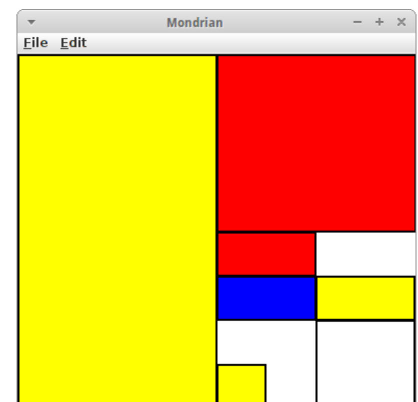
Sierpinski Triangle

Das Sierpinski-Dreieck [8] ist ein anderes visuelles Beispiel für Rekursion. Dabei beginnt man mit dem ganz großen Dreieck, darin zeichnet man dann drei kleine Dreiecke, und in jedes dieser kleinen wieder drei kleine, und so weiter.

```
void drawSierpinski(double x, double y, double w,
double h) {

    drawTriangle(x, y, w, h);

    // base case
    if ((w < 2.0) || (h < 2.0)) {
        return;
    }

    // recursive case
    double h2 = h / 2;
    double w2 = w / 2;
    drawSierpinski(x, y, w2, h2);
    drawSierpinski(x + w2 / 2, y + h2, w2, h2);
    drawSierpinski(x + w2, y, w2, h2);

}
```

Die *drawTriangle()* benutzt eine GPolygon um ein Dreieck beginnend an der Position x,y mit der Breite w und der Höhe h zu zeichnen.

Beim Sierpinski-Dreieck handelt es sich um ein typisches Fraktal. Fraktale haben die Eigenschaft der Selbstähnlichkeit, d.h. sie bestehen aus kleineren Teilen von sich selbst, die sich immer wieder wiederholen. Es gibt übrigens eine interessante Beziehung zwischen dem Tower of Hanoi Problem und dem Sierpinski Dreieck [9], und auch das Pascalsche Dreieck hat mit dem Sierpinski Dreieck verwandschaftliche Beziehungen.



Mondrians

Piet Mondrian [10] war ein niederländischer Maler dessen späteren Werke vor allem der Rekursion gewidmet waren. Bewiesen haben das Eric Roberts und Julie Zelenski [11]. Der Beweis geht wie folgt:

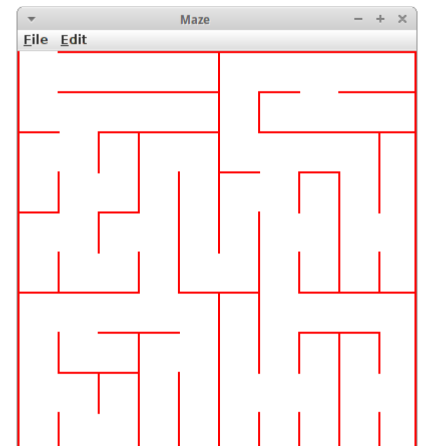
1. im ersten Schritt haben wir drei Optionen:
    1. wir teilen die Leinwand horizontal in zwei kleinere Leinwände oder
    2. wir teilen die Leinwand vertikal in zwei kleinere Leinwände oder
    3. wir tun gar nichts;
2. für jede der kleineren Leinwände wenden wir wieder Schritt 1 an, bis die Leinwände zu klein sind.

In Java sieht das dann so aus:

```
private void drawMondrian(int i, int j, int width, int height) {
    // base case
    if ((width < MIN_SIZE) || (height < MIN_SIZE)) {
        return;
    }

    // recursive case
    int choice = rgen.nextInt(0, 2);
    switch (choice) {
    case 0: // divide canvas horizontally
        drawMondrian(i, j, width / 2, height);
        drawMondrian(i + width / 2, j, width / 2, height);
        break;
    case 1: // divide canvas vertically
        drawMondrian(i, j, width, height / 2);
        drawMondrian(i, j + height / 2, width, height / 2);
        break;
    default: // do nothing
        drawRectangle(i, j, width, height);
        break;
    }

}
```

Dabei zeichnet *drawRectangle()* ein GRect mit einer zufälligen Farbe. Damit das aber dann wie ein Mondrian aussieht, sollte man sich bei den Farben auf weiß, blau, gelb und rot beschränken. Nicht alle Mondrians werden was, man muss das ein paar Mal laufen lassen. Aber ab und zu kommen echte Originale raus.

Maze



Es gibt zig Algorithmen um Labyrinthe zu erzeugen. Einer der einfacheren ist ein rekursiver, der aus einem Rechteck ein Labyrinth erzeugt [12]:

1. wähle einen zufälligen Punkt innerhalb des großen Rechtecks und zeichne zwei Wände durch diesen Punkt, eine horizontal, die andere vertikal, es ergeben sich vier kleine Rechtecke;
2. die beiden Wände die wir gerade eingezogen haben, teilen sich gegenseitig in je zwei Hälften, deswegen haben wir jetzt vier innere Wände. In drei dieser Wände machen wir ein Loch an einer zufälligen Position;
3. für jedes der vier kleinen Rechtecke wiederholen wir Schritt 1 solange bis die Breite oder Höhe der Rechtecke gleich der Zellbreite ist.

Als erstes benötigen wir eine Methode, die unser großes Rechteck mit je zwei Wänden zerteilt:

```
divisionByTwoWalls(0, SIZE, 0, SIZE);
```

In dieser Methode checken wir als erstes den *base case*:

```
private void divisionByTwoWalls(int x0, int x1, int y0, int y1) {
    // base case
    if ((x1 - x0) < 2 * MIN_WIDTH || (y1 - y0) < 2 * MIN_WIDTH) {
        return;
    }

    ...
```

dabei ist *MIN_WIDTH* die Zellbreite. Kommen wir zum *recursive* case. Hier wählen wir einen zufälligen Punkt innerhalb unseres Rechtecks (der aber quantisiert sein sollte):

```
        int x = rgen.nextInt(x0 + MIN_WIDTH, x1 - MIN_WIDTH) / MIN_WIDTH *
    MIN_WIDTH;

        int y = rgen.nextInt(y0 + MIN_WIDTH, y1 - MIN_WIDTH) / MIN_WIDTH *
    MIN_WIDTH;
```

dann sollen wir zwei Wände einzeichnen. Es ist aber besser vier halbe Wände einzuzeichnen:

```
        int noHole = rgen.nextInt(0, 3);
        drawLineWithRandomOpening(x, y0, x, y, noHole == 0);
        drawLineWithRandomOpening(x, y, x, y1, noHole == 1);
        drawLineWithRandomOpening(x0, y, x, y, noHole == 2);

        drawLineWithRandomOpening(x, y, x1, y, noHole == 3);
```

von denen eine kein Loch bekommt. Bleibt nur noch, dass wir uns selbst wieder für jedes der vier kleinen Rechtecke rekursiv aufrufen:

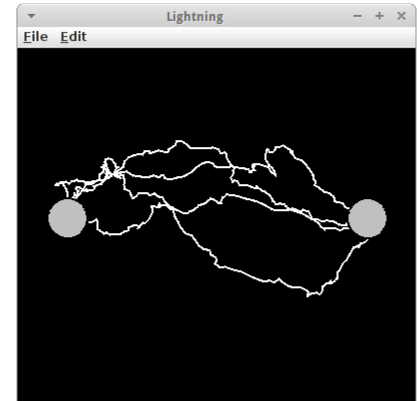
```
        divisionByTwoWalls(x0, x, y0, y);
        divisionByTwoWalls(x, x1, y0, y);
        divisionByTwoWalls(x0, x, y, y1);
        divisionByTwoWalls(x, x1, y, y1);

    }
```

Die Methode *drawLineWithRandomOpening()*, die eine Wand mit einem zufälligen Loch zeichnet sieht wie folgt aus:

```
    private void drawLineWithRandomOpening(int x0, int y0, int x1, int y1,
            boolean withNoOpening) {
        if (withNoOpening) {
            drawLine(x0, y0, x1, y1);
        } else {
            if (x0 == x1) { // vertical
                int y = rgen.nextInt(y0, y1 - MIN_WIDTH) / MIN_WIDTH
                        * MIN_WIDTH;
                drawLine(x0, y0, x1, y);
                drawLine(x0, y + MIN_WIDTH, x1, y1);

            } else {// horizontal
                int x = rgen.nextInt(x0, x1 - MIN_WIDTH) / MIN_WIDTH
                        * MIN_WIDTH;
                drawLine(x0, y0, x, y1);
                drawLine(x + MIN_WIDTH, y0, x1, y1);

            }
        }

    }
```

Eigentlich müsste das ein Fünfzeiler sein, mir ist aber einfach keine einfachere Lösung eingefallen.

Lightning



Im Internet bin ich zufällig über eine interessante Anwendung für Rekursion gestoßen, einen Blitzgenerator [13]. Er verwendet den sogenannten Midpoint-Displacement Algorithmus [14] um Blitze zu zeichnen. Der Mittelpunktverschiebungs-Algorithmus funktioniert folgendermaßen:

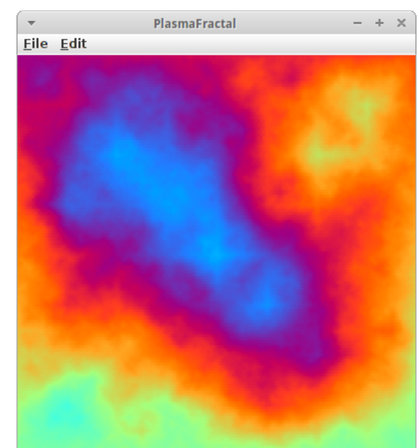
1. wir beginnen mit zwei Punkten, die wir mit einer Linie verbinden wollen;
2. aber anstelle die Linie direkt zu zeichnen, halbieren wir die Strecke, und verschieben den Mittelpunkt der Strecke senkrecht zur Strecke um einen gewissen Betrag, das *Displacement*;
3. mit den beiden Hälften wiederholen wir das Verfahren, dabei wird der Betrag des *Displacements* jedesmal halbiert;
4. wir hören auf, wenn das *Displacement* kleiner als ein bestimmter Wert ist.

In Java sieht das Ganze dann so aus:

```
private void drawLightning(double x1, double y1, double x2, double y2,
        double displace) {
    // base case
    if (displace < 2) {
        drawLine(x1, y1, x2, y2);

    // recursive case
    } else {
        double mid_x = (x2 + x1) / 2.0;
        double mid_y = (y2 + y1) / 2.0;
        mid_x += (Math.random() - 0.5) * displace;
        mid_y += (Math.random() - 0.5) * displace;
        drawLightning(x1, y1, mid_x, mid_y, displace / 2);
        drawLightning(x2, y2, mid_x, mid_y, displace / 2);
    }

}
```

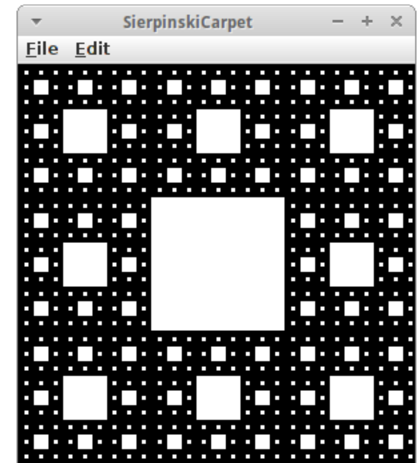
Man könnte es jetzt in unserer CityAtNight aus dem ersten Semester blitzen lassen...

PlasmaFractal



Den Mittelpunktverschiebungs-Algorithmus [14] kann man nicht nur auf Linien, sondern auch auf Ebenen anwenden [15]. Man beginnt mit einem Rechteck, legt die Farben der Eckpunkte fest, und gibt dem Mittelpunkt des Rechtecks eine zufällige Farbe. Man teilt das Rechteck dann in vier kleinere Rechtecke und beginnt mit dem Prozess von Vorne. Das macht man solange, bis die Rechtecke kleiner als ein Pixel sind. Das Verfahren eignet sich sehr gut um Wolken zu generieren, dann sollte man natürlich verschiedene Blautöne als Farben wählen. Man kann es aber auch zur Generierung von Terrains verwenden, dann entsprechen die Farbewerte den Höhen im Terrain.

Challenges

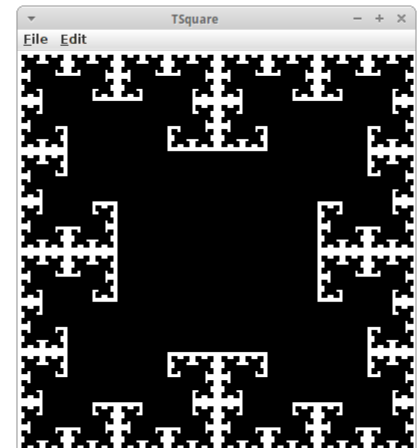


SierpinskiCarpet

Der Herr Sierpinski hat sich nicht nur mit Dreiecken sondern auch mit Teppichen beschäftigt [16]. Es handelt sich auch wieder um eine fraktale Struktur. Um einen Sierpinski Teppich herzustellen folgt man diesen Anweisungen:

1. teile das Rechteck in neun gleich große Rechtecke;
2. entferne das mittlere, und wiederhole den ersten Schritt mit den acht übrig gebliebenen Rechtecken.

Es gibt auch eine dreidimensionale Version des Sierpinski Teppichs, die auch unter dem Namen Menger-Schwamm bekannt ist [17].

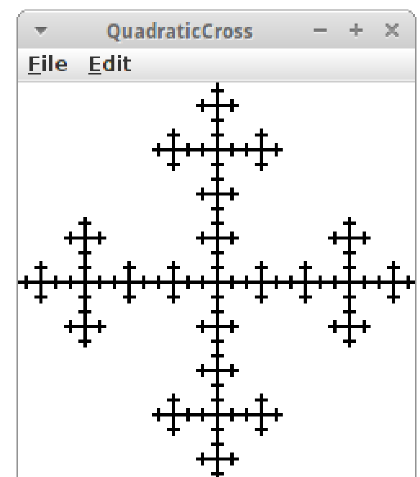


TSquare

Ähnlich wie die Sierpinski Dreiecke und Teppiche ist auch das TSquare ein Fraktal. Die Konstruktion ist eigentlich auch ganz einfach:

1. beginne mit einem Quadrat, das halb so groß ist wie die Fläche die zur Verfügung steht, und platziere es in die Mitte;
2. an jeder Ecke diese Quadrats platziere mittig je ein neues Quadrat, das halb so groß ist wie das ursprüngliche;
3. wiederhole den Vorgang solange, bis die Quadrat die Größe eines Pixels haben.

Man kann so etwas natürlich auch mit Dreiecken, Fünfecken, usw. machen.



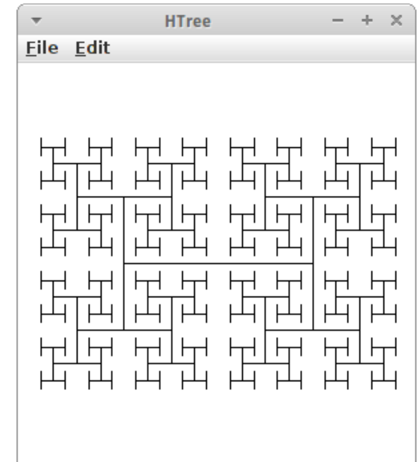
QuadraticCross

Das Quadratic Cross, auch Vicsek Fraktal genannt, ist ähnlich zum TSquare. Aber anstelle Quadrate hinzuzufügen, werden welche weggenommen. Wir beginnen also mit einer schwarzen Fläche. Dann folgen wir diesen Schritten:

1. teile das Quadrat in neun gleich große Quadrate, und entferne die Eckquadrate;
2. tue das gleich mit den übrig gebliebenen fünf Quadraten.

Dabei kommt dann etwas zum Vorschein, dass ein bisschen wie ein Kreuz aussieht. Antennen in Handys werden von diesem Fraktal inspiriert [20].

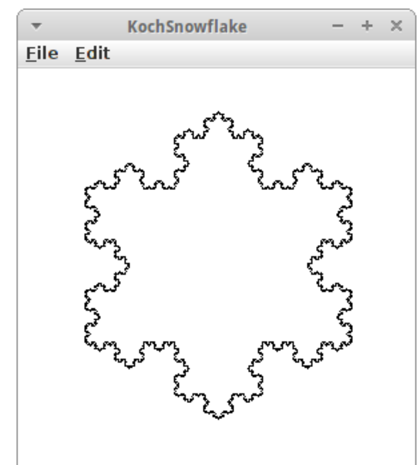
HTree



Später wenn wir mit Bäumen arbeiten, dann werden wir den H-Baum brauchen [21]. Im Moment beschränken wir uns darauf, dass er ganz einfach zu konstruieren ist. Die Konstruktionsvorschrift geht folgendermaßen:

1. zeichne eine Linie in die Mitte, die halb so lange ist wie die Breite die zur Verfügung steht;
2. dann zeichne zwei Linien an den jeweiligen Endpunkten, die senkrecht sind, und deren Länge 1/√2 mal so lange ist;
3. wiederhole Schritt 2 solange bis die Länge kleiner als z.B. 15 ist.

KochSnowflake



Kommen wir zum letzten Projekt in diesem Kapitel, der Koch-Kurve [22], die etwas an eine Schneeflocke erinnert. Auch bei der Koch-Kurve handelt es sich um ein Fraktal. Die Koch-Kurve beginnt als Dreieck:

1. zeichne ein gleichseitiges Dreieck;
2. teile jedes der drei Seitenteile in drei gleiche Teile;
3. entferne das mittlere dieser drei Teile, und ersetze es durch ein gleichseitiges Dreieck das nach außen zeigt;
4. entferne die Basis dieses neuen Dreiecks;
5. wiederhole Schritt 2.

Auf der Seite der Wikipedia gibt es eine Animation in der man in die Koch-Kurve "hineinfliegt", eine sehr schöne Verdeutlichung was Selbstähnlichkeit wirklich bedeutet.

Research

In diesem Kapitel gibt es mal was richtig Interessantes zu erforschen.

Gödel's Unvollständigkeitssatz

Für die Mathematiker ging die Welt unter, den Rest der Menschheit hat es eigentlich nicht interessiert: der Gödelsche Unvollständigkeitssatz [23]. Die traurige Geschichte beginnt mit der *Principia Mathematica* [24] von Russell und Whitehead, führt dann zu *Russell's Paradox* [25], und endet mit *Gödel's Unvollständigkeitssatz*. Wenn einer keine traurigen Geschichten mag, sollte er das vielleicht nicht nachlesen, wen aber Horror pur leidenschaftlich anzieht, dem wird die Geschichte gefallen. Im Ernst, es ist schockierend wie wenige Leute jemals über die Erschütterung (und Zerstörung) der Grundfeste der Mathematik etwas gehört haben. Spricht für unser Bildungssystem...

Fragen

1. Der folgende Code ist ein Beispiel für die Berechnung der Fakultät einer Zahl. Aber etwas stimmt nicht mit diesem Code. Was ist es?

```
public int factorial(int n) {
    return n * factorial( n-1 );

}
```

2. Beschreiben Sie in griben Zügen ein Programm, das einen Baum rekursiv zeichnet.

3. Ist der folgende Algorithmus ein rekursiver oder ein iterativer Algorithmus?

```
private int choose(int n, int k) {
    if ( (k == 0) || (k == n) ) {
        return 1;
    } else {
        return choose(n-1, k) + choose(n-1, k-1);
    }

}
```

4. Was ist ein rekursiver Algorithmus, was sind seine Vorteile, was sind seine Nachteile?

5. Schreiben Sie zwei Methoden namens *powerIteration(double x, int n)* und *powerRecursion(double x, int n)*. Die erste Funktion sollte die Leistung der Zahl x auf die Leistung n iterativ berechnen, die zweite rekursiv.

6. Rekursion folgt immer dem gleichen Muster. Nennen Sie die beiden Schritte die nötig sind, um einen Algorithmus rekursiv umzusetzen.

7. Schreiben Sie eine rekursive Methode public boolean *isPalindrome(String s)* die prüft, ob eine gegebene Zeichenkette ein Palindrom ist. ("otto" und "rentner" sind zwei Beispiele für Palindrome.)

8. In der Vorlesung und auch den Labs haben wir “Mondrians” gezeichnet. Skizzieren (malen) Sie grob wie diese aussehen.

9. In der Vorlesung und auch den Labs haben wir Sierpinski Dreiecke (Sierpinski Triangles) gezeichnet. Skizzieren (malen) Sie grob wie diese aussehen.

10. Was ist ein "divide and conquer" Algorithmus? Erklären Sie, wie diese funktionieren.

Referenzen

Das wunderbare Buch "Einführung in die Programmierung in Java" von Robert Sedgewick und Kevin Wayne hat eine wunderbare Sammlung von Rekursionsalgorithmen: http://www.cs.princeton.edu/introcs/23recursion/. Das ist eigentlich Pflichtlektüre.

[1] Palindrome, https://en.wikipedia.org/wiki/Palindrome

[2] The long and short of it, The Economist, Johnson Language, www.economist.com/blogs/johnson/2010/06/short_and_long_words

[3] Tower of Hanoi, https://en.wikipedia.org/wiki/Tower_of_Hanoi

[4] Permutation, https://en.wikipedia.org/wiki/Permutation

[5] Subset, https://en.wikipedia.org/wiki/Subset

[6] Pascalsches Dreieck, https://de.wikipedia.org/wiki/Pascalsches_Dreieck

[7] Combination, https://en.wikipedia.org/wiki/Combination

[8] Sierpinski triangle, https://en.wikipedia.org/wiki/Sierpinski_triangle

[9] Tower of Hanoi, https://en.wikipedia.org/wiki/Tower_of_Hanoi

[10] Piet Mondrian, https://en.wikipedia.org/wiki/Piet_Mondrian

[11] Programming Abstractions in C++, Eric S. Roberts and Julie Zelenski, Stanford University

[12] Maze generation algorithm, https://en.wikipedia.org/wiki/Maze_generation_algorithm

[13] Lightning Generator, https://krazydad.com/bestiary/bestiary_lightning.html

[14] Diamond-square algorithm, https://en.wikipedia.org/wiki/Diamond-square_algorithm

[15] Plasma Fractal, Justin Seyster, http://jseyster.github.io/plasmafractal/

[16] Sierpinski carpet, https://en.wikipedia.org/wiki/Sierpinski_carpet

[17] Menger sponge, https://en.wikipedia.org/wiki/Menger_sponge

[18] T-square, https://en.wikipedia.org/wiki/T-square_(fractal)

[19] Vicsek fractal, https://en.wikipedia.org/wiki/Vicsek_fractal

[20] Fractal antenna, https://en.wikipedia.org/wiki/Fractal_antenna

[21] H tree, https://en.wikipedia.org/wiki/H_tree

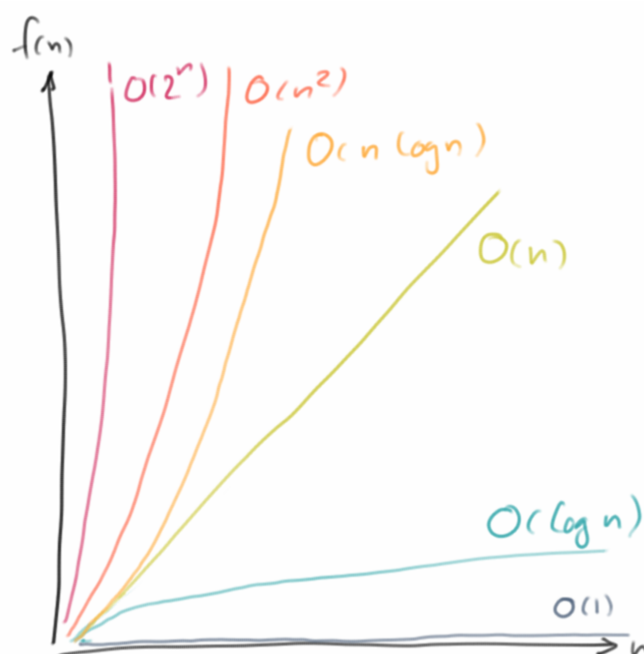
[22] Koch snowflake, https://en.wikipedia.org/wiki/Koch_snowflake

[23] Kurt Gödel, https://en.wikipedia.org/wiki/Kurt_Gödel

[24] Principia Mathematica, https://en.wikipedia.org/wiki/Principia_Mathematica

[25] Russell's paradox, https://en.wikipedia.org/wiki/Russell%27s_paradox

Algorithmic Analysis



Wir haben jetzt schon einige Algorithmen gesehen, und u.a. festgestellt, dass es sehr häufig mehr als einen Algorithmus gibt um ein gegebenes Problem zu lösen. Die Frage stellt sich, welchen soll ich denn verwenden? In diesem Kapitel beschäftigen wir uns damit welcher denn der schnellste ist, und wie man das vorhersagt oder misst. Geschwindigkeit ist aber nicht das einzige Kriterium das für die Wahl entscheidend sein kann: manchmal ist es Speicherplatz, manchmal ist es Zuverlässigkeit, und manchmal Einfachheit. Aber in aller Regel hilft es einem wenig, wenn eine Algorithmus Jahre braucht bis er fertig ist, deswegen ist das häufigste Optimierungskriterium die Zeit. Außerdem stellen wir auf den folgenden Seiten auch ein paar neue Techniken vor: Approximation, Dynamische Programmierung und Divide and Conquer.

Approximation

Als wir Leute in einem Stadium zählen wollten, haben wir das erste Mal gesehen, dass es u.U. genügt nur ungefähr zu wissen wieviel Leute es sind. Also eine grobe *Abschätzung* oder eine Annäherung an den genauen Wert. Das ist eine Technik die nicht selten verwendet wird, vor allem wenn es schnell gehen muss.

Wenn wir die Fakultät einer Zahl berechnen wollen, also *n!*, dann haben wir im letzte Kapitel schon zwei Methoden gesehen, und zwar Iteration und Rekursion. Eine dritte ist Approximation. In der Regel muss man da einen Mathematiker als Kumpel haben, die kennen sich mit so Sachen nämlich gut aus. Für die Fakultät gibt es nämlich eine Formel die sehr gute Näherungswerte für große Fakultäten liefert die sogenannte Stirling-Formel [1]:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Wobei man bei den Mathematikern immer aufpassen muss: was heisst denn groß? Manchmal bedeutet das riesengroß, also total nutzlos eigentlich. Aber hier heisst groß so ab drei, und das ist eigentlich gar nicht so groß. Deswegen ist das eine Superformel um Fakultäten auszurechnen (genauer anzunähern) sobald n größer als drei ist.

Dynamic Programming

Kommen wir zur Technik der *Dynamischen Programmierung* [2,3]. Der Name ist wohl der dümmste der einem einfallen kann, denn die Technik ist eigentlich nicht besonders dynamisch, noch hat sie irgendetwas mit Programmierung zu tun. Wenn man unbedingt will, dann wäre dynamische Nachschlagetabelle, also *dynamic lookup table*, schon beschreibender.

Worum geht es? Wenn wir die Fakultät von 11 berechnen wollen, dann brauchen wir ja die Fakultät von 10 erstmal, weil ja

```
11! = 11 * 10!
```

So was wäre wenn wir irgendwann schon mal 10! ausgerechnet hätten und uns das gemerkt hätten? Das ist Dynamischen Programmierung: einfach merken was man schon mal ausgerechnet hat. Oder genauer, das wiederverwenden was man schon mal ausgerechnet hat.

Schauen wir uns mal an wie man das in Java macht. Das "Merken" machen wir in einem Array:

```
public long[] factorialLookupTable;
```

Dann berechnen wir einfach mal die ersten zwanzig Fakultäten im Voraus:

```
public void initFactorialLookupTable() {
    factorialLookupTable = new long[20+1];
    for (int i = 1; i < factorialLookupTable.length; i++) {
        factorialLookupTable[i] = factorialIterative(i);
    }

}
```

Wenn uns jetzt irgendjemand nach einer Fakultät fragt, schauen wir einfach in unserem Array nach:

```
public long factorialLookupTable(int n) {
    return factorialLookupTable[n];

}
```

und das geht super-schnell. Technisch gesehen schummeln wir hier ein bisschen, weil wir die ganzen Werte im Voraus berechnen. Aber das Kernprinzip der Dynamischen Programmierung ist es das wieder zu verwenden was wir schon ausgerechnet haben.

Fastest

Wir haben ja schon angedeutet, dass es in diesem Kapitel um Geschwindigkeit geht und wie man sie misst. Wir haben inzwischen schon vier verschiedene Algorithmen um die Fakultät einer Zahl zu berechnen:

- Iteration
- Rekursion
- Dynamischen Programmierung
- Abschätzung

Welcher ist nun der schnellste? Zeitmessungen in Java machen wir mit der *System.currentTimeMillis()* Methode, die gibt uns die Zeit die seit dem 01.01.1970 vergangen ist in Millisekunden. Wir nehmen also die Zeit bevor wir mit unserer Berechnung beginnen und ziehen dann die Zeit danach davon ab:

```
long start = System.currentTimeMillis();
for (int i = 0; i < NR_OF_ITERATIONS; i++) {
    x = Factorial.factorialIterative(20);
}
long duration = System.currentTimeMillis() - start;

System.out.println("time iterative: " + duration);
```

Wenn eine Berechnung sehr kurz ist, also im Milli- oder Nanosekunden Bereich, dann sollten wir diese Berechnung mehrmals (so hundertmillionenmal) ausführen, sonst sind die gewonnen Resultate nicht besonders aussagekräftig. Überhaupt kann bei Zeitmessungen recht viel schief gehen.

Und, wer ist der Gewinner?

```
iteration:             1262ms
recursion:             2524ms
dynamic programming:      4ms
approximation:            6ms
```

Der Unterschied ist schon ziemlich massiv: Dynamischen Programmierung und Abschätzung schlagen die Iteration und Rekusion um Längen! Analysieren wir die Ergebnisse aber etwas genauer:

- Abschätzung ist zwar schnell aber nicht genau. Man muss schon wissen ob man mit einer Abschätzung leben kann, aber sie ist schnell und braucht wenig Speicher.
- Dynamischen Programmierung ist genau und schnell. Aber auch hier gibt es zu bedenken: einmal muss ich die Nachschlagtabelle ja ausrechnen, und das dauert Zeit. Und die Nachschlagtabelle benötigt Speicher, teilweise sehr viel Speicher.
- Iteration ist zwar nicht so schnell, ist aber sehr effektiv was den Speicher angeht. Und sie ist immer noch doppelt so schnell wie die Rekursion (aber nicht immer).
- Rekursion ist eigentlich so das schlimmste was man machen kann: sie ist langsam und braucht viel Speicher. Sehr häufig sind rekursive Lösungen aber sehr elegant, und bekanntlich ist Eleganz ja teuer. Man fragt sich natürlich warum wir uns dann das ganze letzte Kapitel damit rumgeschlagen haben... die Antwort kommt gleich.

Zusammenfassend kann man allerdings sagen, wenn ein Algorithmus schnell ist, dann geht er meist nicht sehr effektiv mit Speicher um, und umgekehrt.

Divide and Conquer

Ist Rekursion immer am langsamsten? Meistens ja, aber es gibt Ausnahmen. Schauen wir uns mal das Berechnen von Potenzen an. Nehmen wir an wir wollen zwei hoch acht berechnen:

```
28 = 2*2*2*2*2*2*2*2 = 256
```

Eine Möglichkeit ist mittels Iteration:

```
int powerIt(int x, int n) {
    int power = 1;
    for (int i = 0; i < n; i++) {
        power *= x;
    }
    return power;

}
```

Eine zweite ist mittels Rekursion. Erinnern wir uns, dass

$2^8 = 2 * 2^7$

dann sehen wir hier sofort eine rekursive Beziehung. Mit dem *base case* dass $2^0 = 1$, können wir das folgendermaßen in Java umsetzen:

```
int powerRe(int x, int n) {
    if (n == 0)
        return 1;
    else
        return x * powerRe(x, n-1);

}
```

Interessanterweise ist das aber nicht die einzige rekursive Lösung. Es gibt nämlich noch eine andere:

$2^8 = 2^4 * 2^4$

Das sieht jetzt ganz unscheinbar aus, ist es aber nicht. Es ist das erste Mal dass wir einem Algorithmus aus der Kategorie *divide and conquer* begegnen. Die haben's in sich. Zunächst setzen wir das in Java um:

```
int powerDC(int x, int n) {
    if (n == 0)
        return 1;
    else {
        int temp = powerDC(x, n/2);
        if (n % 2 == 0)
            return temp * temp;
        else
            return x * temp * temp;
    }

}
```

Jetzt stellt sich natürlich wieder die Frage, welcher ist denn der schnellste? Wir lassen unsere Stoppuhren laufen und stellen fest:

```
iteration:              6ms
recursion:            778ms
divide & conquer:      89ms
approximation:          3ms
```

Wie erwartet, ist die Approximation die schnellste. Iteration ist auch verflucht schnell, aber die Überraschung kommt wenn wir Rekusion mit Divide and Conquer vergleichen: Divide and Conquer ist fast zehnmal schneller! Es kommt sogar noch besser: je größer die Zahlen werden, desto besser wird Divide and Conquer, am Ende schlägt er sogar die Iteration. Wie wir in späteren Kapiteln noch sehen werden, Divide and Conquer ist unser Freund.

Algorithm Analysis

Wie bewertet man denn einen Algorithmus? Bisher haben wir einfach die Stoppuhr ausgepackt und gemessen. Das funktioniert, aber es ist nicht sehr wissenschaftlich, und wir wissen eigentlich auch nicht warum ein Algorithmus besser ist als ein anderer. Darum geht es in der algorithmischen Analyse, man versucht Algorithmen mathematisch zu untersuchen, zu bewerten und einzuordnen. Da wir keine ausgebildeten Mathematiker sind, werden wir das ganze etwas vereinfachen: im Prinzip was wir im Folgenden machen ist nichts anderes als zählen.

Constant Performance: O(1)

Wir beginnen ganz einfach, wir nutzen die Tatsache, dass die meisten modernen CPUs die Potenz einer Zahl direkt berechnen können:

```
int powerApprox(int x, int n) {
    return (int) Math.pow(x, n);

}
```

Dies dauert eine feste Anzahl von CPU-Zyklen (etwa 28 CPU-Zyklen) und egal, was die Werte von x oder n sind, wird es immer die gleiche Zeit in Anspruch nehmen. Wir nennen dies konstante Performanz, weil sich die Zeit nicht ändert, auch wenn wir x oder n ändern. In Kurzform: O(1).

Linear Performance, Iteration: O(n)

Als nächstes betrachten wir unsere iterative Lösung:

```
1.  int powerIt(int x, int n) {
2.      int power = 1;
3.      for (int i = 0; i < n; i++) {
4.          power *= x;
5.      }
6.      return power;

7.  }
```

Wir wollen ermitteln, wieviele CPU-Zyklen für diese Berechnung notwendig sind. Bei modernen CPUs dauern die meisten Anweisungen etwa 2 bis 3 Zyklen. Damit kann man dann ausrechnen wie lange es dauert. Deshalb genügt es für uns eigentlich zu wissen wieviele Anweisung für eine gewisse Berechnung nötig sind. Also analysieren wir den Code oben Zeile für Zeile:

1. die Werte von x und n müssen wir in den CPU-Registern speichern, das benötigt 2 Anweisungen;
2. eine einfache Zuweisung, braucht 1 Anweisung;
3. Schleifen sind etwas knifflig:
    1. einmal haben wir eine Zuordnung, int i = 0, das ist 1 Anweisung, wird nur einmal ausgeführt;
    2. dann haben wir einen Vergleich, i < n, auch 1 Anweisung, wird u.U. mehrmal ausgeführt;
    3. schließlich haben wir noch ein Inkrement, i++, auch 1 Anweisung, wird u.U. mehrmal ausgeführt;
4. eine einfache Multiplikation, dauert 1 Anweisung;
5. eine geschloßene geschweifte Klammer benötigt natürlich keine Zeit, aber wenn die Schleife mehrmals durchlaufen wird, bedeutet sie, dass wir wieder zurück zur Zeile 3 gehen;
6. return heißt soviel wie "schreibe es irgendwo in den Speicher", d.h. 1 Anweisung.

Also, wielange dauert es? Dass hängt von *n* ab. Je größer *n* ist, desto länger dauert es. Um genau zu sein

```
# of instructions = 2+1+1 + n*( 1+1+1) + 1 = 5 + 3*n ~ n
```

D.h. je größer *n* wird, desto länger dauert es. Man nennt das auch lineare Performanz, linear in *n*. Oder in Kurzform: O(n).

Linear Performance, Recursion: O(n)

Auch Rekursion kann lineare Performanz haben. Unsere erste rekursive Lösung war wie folgt:

```
1.  int powerRe(int x, int n) {
2.      if (n == 0)
3.          return 1;
4.      else
5.          return x * powerRe(x, n-1);
6.  }
```

Wir analysieren wieder Zeile für Zeile:

1. wir müssen x und n in den CPU-Registern speichern, das sind 2 Anweisungen;
2. ein einfacher Vergleich, 1 Anweisung;
3. return schreibt etwas in den Speicher, d.h. 1 Anweisung;
4. *else* zählt nicht als Anweisung, ist Teil von *if*, also 0 Anweisungen;
5. das ist jetzt wieder etwas komplizierter, gehen wir von rechts nach links vor:
    1. wir subtrahieren 1 von n, 1 Anweisung;
    2. wir rufen uns selbst auf: keine Ahnung wie lange das dauert, also ? Anweisungen
    3. eine Multiplikation, 1 Anweisung;
    4. wir geben etwas zurück, das ist 1 Anweisung.

Heiklig ist der rekursive Aufruf. Wie kann man den abschätzen? Um ein Gefühl dafür zu bekommen was passiert, wählen wir einfach mal verschiedene *n*, beginnend mit n = 0:

- **n=0:** es gibt gar keinen rekursiven Anruf, wir kommen nur zur Zeile 3, macht insgesamt 4 Anweisungen;
- **n=1:** hier gibt es einen rekursiven Anruf, der benötigt 4 Anweisungen (d.h. n = 0), womit wir insgesamt bei 2 + 1 +1 + 4 + 1 + 1 = 4 + 6 = 10 Anweisungen wären;
- **n=2:** hier gibt es einen rekursiven Aufruf zu n = 1, der 9 Anweisungen benötigt, wie wir gerades festgestellt haben, also in Summe 2 + 1 + 1 + 9 + 1 + 1 = 9 + 6 = 15 Anweisungen.

Wie wir sehen, fügen wir mit jedem weiteren Schritt weitere 6 Anweisungen hinzu. Damit können wir die Anzahl der Anweisungen für ein beliebiges *n* vorhersagen:

```
# of instructions = 4 + 6*n ~ n
```

Das sieht fast genauso aus wie bei der Iteration, nur mit einem anderen Faktor. Wieder, je größere *n* wird, desto länger dauert es. Auch hier haben wir es mit linearer Performanz zu tun. In Kurzform: O(n).

Logarithmic Performance, Divide and Conquer: O(log(n))

Kommen wir zu unserem zweiten rekursiven Ansatz, der *Divide and Conquer* Lösung:

```
1.  int powerDC(int x, int n) {
2.      if (n == 0)
3.          return 1;
4.      else {
5.          int temp = powerDC(x, n/2);
6.          return temp * temp;
7.      }
8.  }
```

Wir haben eine etwas einfachere Version des Algorithmus ausgewählt, die nur für Potenzen von 2 funktioniert, also n = 1, 2, 4, 8, 16, ... Wir könnten auch die genaue Lösung von vorher nehmen, aber es wäre ein bisschen komplizierter. Wir analysieren wieder jede Zeile:

1. wir speichern wieder x und n in den Registern der CPU, also 2 Anweisungen;
2. ein einfacher Vergleich, 1 Anweisung;
3. return speichert in den Speicher, d.h., 1 Anweisung;

4. *else* zählt wieder nicht, 0 Anweisungen;
5. hier wird es wieder etwas komplizierter:
    1. erst mal dividieren wir durch 2, bei Ganzzahlen ist das ein einfacher Shift, also 1 Anweisung;
    2. dann rufen wir uns selbst auf: hängt von *n* ab wie lange das dauert, ? Anweisungen;
    3. und zu letzt, noch ein Zuweisung, macht 1 Anweisung;
6. hier haben wir eine Multiplikation und ein *return* Statement, sollten 2 Anweisungen sein.

Wie vorher ist auch hier der rekursive Aufruf den wir uns genauer ansehen müssen. Wie oben, nehmen wir einfach mal ein paar verschiedene *n*, beginnend mit 0:

- **n=0:** es gibt keinen rekursiven Anruf, wir kommen nur zur Zeile 3, also insgesamt 4 Anweisungen;
- **n=1:** es gibt einen rekursiven Anruf, der 4 Befehle benötigt (d.h. n=0), in Summe also: 2 + 1 + 1 + 4 + 1 + 2 = 4 + 7 = 11 Anweisungen;
- **n=2:** es gibt einen rekursiven Aufruf zu n=2, der 11 Befehle benötigt, in Summe also: 2+1+1+11+1+2 = 11 + 7 = 18 Anweisungen;
- **n=4:** es gibt einen rekursiven Aufruf zu n=4, der 18 Befehle benötigt, in Summe also: 2+1+1+18+1+2 = 18 + 7 = 25 Anweisungen.

Also keine große Sache, oder? Vergleichen wir mal Äpfel mit Birnen, soll heißen, vergleichen wir mal den einfachen rekursiven Ansatz mit der Divide and Conquer Lösung:

log_2(n)	n	recursion	divide & conquer
0	1	9	11
1	2	14	18
2	4	24	25
3	8	44	32
4	16	84	39
5	32	164	46
6	64	324	53

Solange *n* klein ist, ist kaum ein Unterschied festzustellen. Aber sobald *n* ein bisschen größer wird, z.B. 64, dann beginnt man einen Unterschied festzustellen. Und je größer *n* wird, desto größer wird auch der Unterschied. Wir können ganz einfach sehen, dass für den Divide and Conquer Algorithmus die Anzahl der Anweisungen gegeben ist durch

```
# of instructions = 11 + 7* Log_2( n )
```

Dabei ist *log_2()* der Logarithmus zu Basis zwei. Dies ist das erste Mal, dass wir einem Algorithmus begegnen der logarithmisch in seiner Performanz ist, in Kurzform: O(log(n)). Logarithmisch ist gut, weil es schnell bedeutet!

Comparing Algorithms and Big-O Notation

Für die vier Algorithmen die wir bisher genauer analysiert haben, haben wir folgendes herausgefunden:

	# of instructions	Big-O
powerApprox()	28	O(1)
powerIt()	5 + 3 * n	O(n)
powerRe()	4 + 5 * n	O(n)
powerDC()	4 + 7 * log(n)	O(log(n))

Wenn die *n* sehr groß werden, dann spielen konstante Faktoren eigentlich nur eine sehr geringe Rolle, weswegen wir sie vernachlässigen. Wir benutzen dann etwas, das wir als Big-O Notation bezeichnen: uns interessiert nur wie das Laufzeitverhalten, die Performanz, von *n* abhängt. Noch genauer, uns interessiert eigentlich nur das Verhalten für den schlimmsten Fall, deswegen ignorieren wir alle Faktoren die für große *n* unwichtig werden:

```
Time  =  2 * n + 4      ->  O(n)
Time  =  4 * n - 1      ->  O(n)
Time  =  1/4 * n² - n   ->  O(n²)
Time  =  2n + n²        ->  O(2ⁿ)
```

Der Grund warum Big-O so nützlich ist, hat damit zu tun, dass es uns erlaubt abzuschätzen wie lange eine Berechnung dauern wird.

Nehmen wir an wir hätten drei Algorithmen, einen der linear ist, O(n), einen der quadratisch ist, $O(n^2)$, und einen der exponentiell ist, $O(2^n)$. Bei ersten Tests konnten wir messen, dass die Berechnung für n=1 ungefähr 1 Millisekunde gedauet hat. Die Frage ist wie lange wird es wohl dauern für n=100?

- **O(n):** da der Algorithmus linear ist, wird er proportional zu n sein, d.h. er wird ca. 100 Millisekunden benötigen, das ist eine Zehntelsekunde. Das ist ok.
- **$O(n^2)$:** der Algorithmus ist quadratisch, d.h., wir müssen die n=100 quadrieren, also 10000, d.h. es dauert ca. 10 Sekunden, das ist auch noch o.k.
- **$O(2^n)$:** der Algorithmus ist exponentiell, d.h., wir müssen 2 hoch 100 nehmen, was in etwa 10^{30} Millisekunden oder 10^{27} Sekunden sind.

Ist das viel oder wenig? Nun das Alter des Universum ist ungefähr 10^{17} Sekunden. Wir werden also das Ende der Rechnung nicht mehr erleben.

Best-Worst-Average Case

Häufig hängt das Laufzeitverhalten eines Algorthmus von der Eingabe ab. Als Beispiel betrachten wir die Suche in einem Array von Strings:

```
boolean search(String[] names, String key) {
    for (int i=0; i < names.length; i++) {
        if (names[i] == key) return true;
    }
    return false;
}
```

Wie lange dauert es, wenn der gesuchte String am Anfang, in der Mitte, am Ende oder vielleicht gar nicht im Array ist?

- Best case:
  Wenn der gesuchte String am Anfang ist, dann ist das super-schnell. Ein Vergleich und wir sind fertig.

- Worst case:
  Ist der gesuchte String am Ende oder vielleicht gar nicht im Array, dann dauert das am längsten: wir müssen durch das gesamte Array suchen. Das ist also der schlimmste Fall.

- Average case:
  Der gesuchte String ist irgendwo in der Mitte. Das ist der Durchschnittsfall. Der ist allerdings sehr häufig gar nicht so einfach zu berechnen.

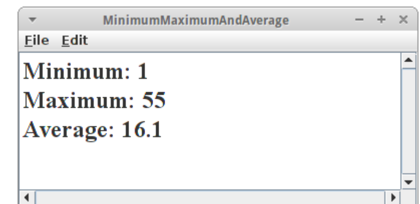
Deswegen interessiert uns eigentlich in der Regel der schlimmste Fall, und das ist was uns die Big-O Notation gibt.

Review

In diesem Kapitel haben wir mehrere neue Konzepte kennengelernt, dazu gehören Approximation, Dynamische Programmierung und der Divide and Conquer Ansatz. Im Allgemeinen gilt, dass es sehr häufig mehrere Algorithmen gibt ein bestimmtes Problem zu lösen. Wir haben gesehen, wie man die Performanz eines Algorithmus messen kann, aber auch wie man mittels detailierter Analyse auch die Performanz eines Algorithmus vorhersagen kann. Dabei war aber auch wichtig festzuhalten, dass nicht immer Geschwindigkeit das Kriterium sein muss nach dem man einen bestimmten Algorithmus auswählt.

Projekte

In den Projekten wollen wir das jetzt ein bisschen durch Beispiele vertiefen. Wir sollten dabei auch ein Gefühl dafür bekommen welches Laufzeitverhalten, O(...), vertretbar ist, und welches für alle praktischen Anwendungen nutzlos ist, weil es einfach zu lange dauert.



Minimum, Maximum and Average

Nehmen wir an wir haben ein Array mit den folgenden Ganzzahlen gegeben:

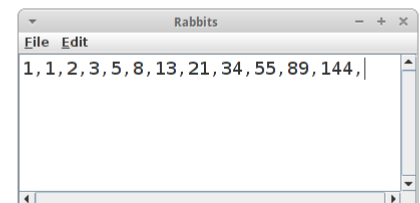
```
int[] arrOfInts =
    { 5, 55, 2, 7, 45, 3, 1, 8, 23, 12 };
```

Wir wollen nun drei Methoden schreiben,

1. eine, die das kleinste Element findet,
2. eine, die das größte Element findet,
3. und eine Methode die den Durchschnitt berechnet.

Dabei wollen wir das Laufzeitverhalten dieser drei Algorithmen bestimmen, entweder über Messungen oder über eine algorithmische Analyse wie wir es oben getan haben.

Die Frage die sich stellt, ist der Algorithmus den wir gefunden haben der einzige oder gibt es auch andere? Und ist er der schnellste? Was wäre denn wenn unser Array sortiert wäre? Könnten wir dann einen anderen Algorithmus finden, und wie wäre sein Laufzeitverhalten?



Rabbits

Jeder der schon mal Kaninchen hatte weiß, dass diese eine interessante Eigenschaft haben: sie vermehren sich und zwar rasant. Wir wollen also ein Programm schreiben welches berechnet wie sich unsere Kaninchen-Population über die Monate entwickelt. Wir folgen dazu dem Modell von Fibonacci [6]:

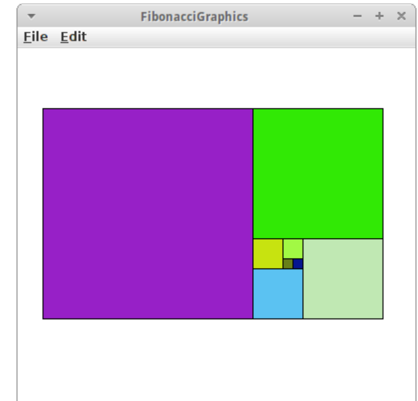
- "Jedes Paar Kaninchen wirft pro Monat ein weiteres Paar Kaninchen.
- Ein neugeborenes Paar bekommt erst im zweiten Lebensmonat Nachwuchs (die Austragungszeit reicht von einem Monat in den nächsten).
- Die Tiere befinden sich in einem abgeschlossenen Raum, sodass kein Tier die Population verlassen und keines von außen hinzukommen kann."

Wenn wir die Simulation richtig schreiben, dann müsste dabei die Fibonacci-Folge, also

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

herauskommen. Interessant ist vielleicht zu beobachten, dass jede Zahl die Summe ihrer zwei Vorgängerzahlen ist.

FibonacciGraphics



Die Fibonacci Zahlen kann man sehr schön visualisieren, eine davon ist die Fibonacci-Spirale [6]. Schauen wir uns die Zahlen noch einmal an:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

Wir wollen daraus ein Kachelmuster aus Quadraten generieren, deren Kantenlängen den Fibonacci Zahlen entspricht. Wir gehen wie folgt vor:

1. lege in die Mitte das ersten Quadrat, rechts daneben legen wir das zweite Quadrat;
2. dann machen wir eine 90 Grad Drehung gegen den Uhrzeigersinn, und legen dort das nächste Quadrat hin;
3. wir wiederholen Schritt 2.

Obwohl der Algorithmus total trivial aussieht, kann man bei der Umsetzung verzweifeln. Man darf sich die Lösung anschauen!

Fibonacci

Bisher haben wir nur Fakultät und Potenzen ausführlich behandelt. Ein anderes sehr schönes Beispiel das mit vielen unterschiedlichen Algorithmen gelöst werden kann ist die Berechnung der Fibonacci-Folge.

Iteration

Für den iterativen Algorithmus muss man lediglich wissen, dass jede Zahl die Summe ihrer zwei Vorgängerzahlen ist. Man beginnt dann einfach mit den ersten beiden, die ja bekannt sind, und berechnet dann eine nach der anderen, bis man diejenige hat die gewünscht war.

Recursion

Die rekursive Version ist sehr elegant:

```
public static long fibonacciRecursive(int n) {
    switch (n) {
    case 0:
        return 0;
    case 1:
        return 1;
    default:
        return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
    }

}
```

aber auch ätzend langsam. Sobald n größer als 40 wird ist die rekursive Version nicht zu gebrauchen.

Divide and Conquer

Es gibt auch eine Divide and Conquer Version zur Berechnung der Fibonacci-Folge, die kann man bei Referenz [7] auf Seite 457 finden.

Dynamic Programming

Die Fibonacci-Folge ist eigentlich eine sehr schöne Anwendung um Dynamische Programmierung richtig zu machen. Sie basiert auf der rekursiven Version, aber anstelle immer und immer wieder die gleichen Zahlen auszurechnen, merkt sie sich wenn sie etwas schon mal ausgerechnet hat:

```
public static long[] fibonacciDynamicProgrammingTable = new long[91 + 1];

private static long fibonacciDynamicProgramming(int n) {
    if (n == 0) {
        return 0;
```

```
        } else if (fibonacciDynamicProgrammingTable[n] > 0) {
            // we have done this calculation before
            return fibonacciDynamicProgrammingTable[n];
        } else {
            long result = 1;
            switch (n) {
            case 1:
                break;
            default:
                result = fibonacciDynamicProgramming(n - 1)
                        + fibonacciDynamicProgramming(n - 2);
            }
            // remember for future use, in case we need it again
            fibonacciDynamicProgrammingTable[n] = result;
            return result;
        }

    }
```

Diese Version ist extrem schnell.

Lookup Table

Klar Lookup Tables haben wir schon mal gesehen. Aber in diesem Beispiel sehen wir auch den Unterschied zwischen Lookup Tables und Dynamischer Programmierung: Bei Lookup Tables werden alle Werte vorausberechnet. Bei Dynamischer Programmierung werden nur die Werte berechnet die wirklich benötigt werden.

Approximation

Auch für die Fibonacci Zahlen gibt es eine Näherungsformel [6]:

$$F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$$

wobei

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.61803\,39887\cdots$$

die *Goldene Zahl* ist.

Results

Hier sind meine Resultate:

```
time iterative:           268ms
time divide and conquer: 2418ms
time dynamic programming:   3ms
time lookup table:          3ms
time approximation:        11ms
```

Interessant ist hier die Dynamische Programmierung: sie ist super-schnell, aber sie hat die Eleganz des rekursiven Algorithmus, macht aber keine unnötigen Berechnungen so wie der Lookup Table Algorithmus, wo ja alles im Voraus berechnet werden muss. (Direkte Rekursion ist nicht dabei, weil sie zu lange dauert, das Buch muss ja irgendwann in den Druck).

Exponential vs. Factorial Time

Bisher haben wir uns eigentlich nur mit "schnellen" Algorithmen beschäftigt, also solchen deren Laufzeitverhalten O(n), O(log(n)) oder O(1) waren. Jetzt wollen wir uns mal kurz mit Algorithmen beschäftigen, deren Laufzeitverhalten eher schlecht ist. Im nächsten Kapitel sehen wir welche mit $O(n^2)$ deswegen beschränken wir uns hier auf die mit exponentiellem Laufzeitverhalten oder noch schlimmer faktoriellem.

Im letzten Kapitel haben wir vier interessante rekursive Algorithmen kennen gelernt:

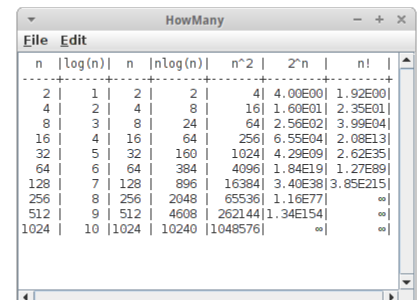
- Subsets,
- Combinationen,
- Permutationen
- und das Tower of Hanoi Problem.

Die Frage die wir hier beantworten wollen, wie ist ihr Laufzeitverhalten? Es gibt zwei Möglichkeiten das zu tun:

1. den Code für verschieden große Inputs ausführen und die Zeit zu messen;
2. eine asymptotische Analyse wie oben ausführen, in der man für verschiedene n zählt oder abschätzt wieviele Anweisungen notwendig sein werden.

Viel Spaß!

Challenges



n	log(n)	n	nlog(n)	n^2	2^n	n!
2	1	2	2	4	4.00E00	1.92E00
4	2	4	8	16	1.60E01	2.35E01
8	3	8	24	64	2.56E02	3.99E04
16	4	16	64	256	6.55E04	2.08E13
32	5	32	160	1024	4.29E09	2.62E35
64	6	64	384	4096	1.84E19	1.27E89
128	7	128	896	16384	3.40E38	3.85E215
256	8	256	2048	65536	1.16E77	∞
512	9	512	4608	262144	1.34E154	∞
1024	10	1024	10240	1048576	∞	∞

How Many

Bevor wir mit dieser Übung beginnen, sollten wir uns erst einmal ausrechnen wieviele Sekunden hat ein Tag, hat ein Jahr, macht ein durchschnttliches Menschenleben aus, und wie alt ist das Universum in Sekunden.

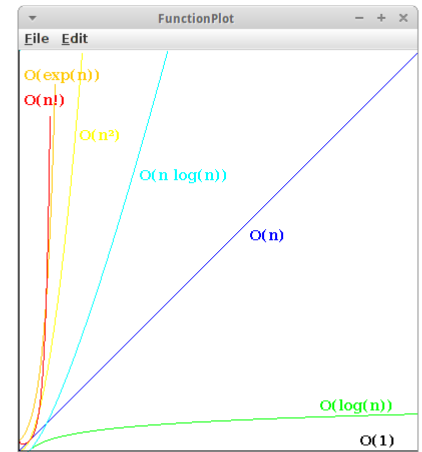
In der Übung geht es darum einfach mal die Werte der mathematischen Funktionen

- log(n)
- n
- n * log(n)
- n^2
- 2^n
- n!

für verschiedene *n* tabellarisch auszugeben. Dabei wollen wir uns aber auf die folgenden *n* beschränken: 2, 4, 8, 16, 32, 64, 128, 256, 512 und 1024. Das Ganze machen wir mit einem ConsoleProgram. (log(n) ist hier der Logarithmus zur Basis zwei).

Wenn wir jetzt einfach mal annehmen, dass die Zahlen in der Tabelle proportional zur Anzahl der CPU Zyklen sind, und ein CPU Zyklus typisch im 100 Nanosekunden (10^{-7} Sekunden) Bereich liegt, können wir abschätzen wielange die entsprechenden Algorithmen benötigen würden um fertig zu werden.

FunctionPlot



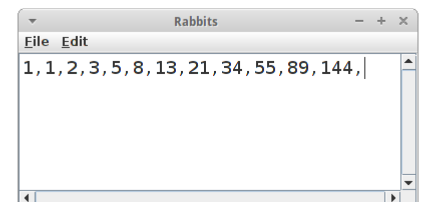
Vielleicht noch besser als die tabellarische Darstellung ist die graphische. Die meiste Arbeit haben wir in der vorhergehenden Übung schon gemacht, jetzt geht es nur noch darum das in ein GraphicsProgram zu verpacken. Dabei lassen wir die x-Werte von 0.1 bis 40 variieren, berechnen die jeweiligen y-Werte für die verschiedenen Funktionen von oben, und zeichnen dieses. Z.B. die Methode die die lineare Funktion zeichnet, könnte wie folgt aussehen:

```
private void plotLinear() {
    double x0 = 0;
    double y0 = SIZE;
    for (double x = 0; x < SIZE; x++) {
        double x1 = 0 + x * SCALE;
        double y = (SIZE - x1 / SCALE);
        if (y < 0)
            break;
        drawLine(x0, y0, x, y, Color.BLUE);
        x0 = x;
        y0 = y;
    }

}
```

dabei zeichnet die Methode *drawLine()* einfach eine GLine zwischen zwei Punkten:

```
private void drawLine(double x0, double y0, double x, double y, Color
col) {
    GLine line = new GLine(x0, y0, x, y);
    line.setColor(col);
    add(line);

}
```

Lookup Table



Wenn wir den Sinus einer Zahl berechnen wollen verwenden wir normalerweise die Taylorreihen Entwicklung [4]:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

wobei hier x im Bogenmaß ist.

Als erstes wollen wir eine Methode namens *sineTaylor(double x)* schreiben die den Sinus einer beliebigen Zahl mittels der Taylorreihen Entwicklung berechnet.

Als zweites, benutzen wir die in Java gegebenen Methode *Math.sin()* zum Vergleich, ob unsere Methode auch taugt.

Als drittes, verwenden wir Lookup-Tables, also Nachschlagetabellen. Dazu schreiben wir eine Methode *sineLookup(double x)*, die eine vorher berechnete Lookup-Table verwendet um den Sinus "zu berechnen". Übrigens, moderne GPUs verwenden genau diesen Ansatz [5].

Jetzt ist die Frage, welcher der drei Ansätze ist der schnellste? Wir können hier auch wieder einfach Zeitmessungen machen, oder wir können eine algorithmische Analyse durchführen. Dieses mal machen wir ein Zeitmessung.

Careful!

Eine kurze Anmerkung: wenn man Performanztests macht, muss man immer sehr vorsichtig sein nicht einer Compiler-Optimierung zum Opfer zu fallen. Der Test könnte erst mal naiv so aussehen:

```
long start = System.currentTimeMillis();
for (int i = 0; i < NR_OF_ITERATIONS; i++) {
    int x = Math.sin(1.2);
}
long duration = System.currentTimeMillis() - start;

System.out.println("time Math.sin(): " + duration);
```

Ganz kritisch ist dabei was genau in der Schleife steht. Wenn wir das so schreiben wie oben, erkennt der Compiler nämlich dass *x* eine lokale Variable ist, und dass danach gar nichts mehr mit ihr passiert. D.h. er ist schlau genug zu wissen, dass er *x* gar nicht ausrechnen muss. Deswegen macht er es auch nicht.

Deswegen rüsten wir auf, und deklarieren x als Instanz- oder Klassenvariable:

```
x = Math.sin(1.2);
```

Jetzt kann der Compiler nicht wissen ob wir evtl. *x* irgendwo anders brauchen, und muss es daher ausrechnen. Aber der Compiler ist immer noch schlauer als wir: er sieht nämlich, dass 1.2 eine Konstante ist. Also warum soll er das denn zig-millionenmal ausrechnen, kommt ja doch immer dasselbe raus. Deswegen müssen wir noch einen drauflegen:

```
x = Math.sin(Math.random());
```

Erst jetzt kann man mit den Zeiten die rauskommen etwas anfangen. Und da stellt sich folgendes heraus:

```
time Math.sin():    6473ms
time sineTaylor(): 3324ms
time sineLookup(): 2476ms
```

Unsere selbstgeschriebe *sineTaylor()* Methode ist ungefähr doppelt so schnell wie die Java native (unsere ist aber viel ungenauer) und die *sineLookup()* Methode ist noch mal 30% schneller, allerdings auch ungenauer. Man kann mit ein paar Tricks die Genauigkeit der Lookup Methode aber so erhöhen, dass es sich einfach nicht rentiert Taylor loszuschicken [5].

Research

Diesem Kapitel war ziemlich anstrengend, deswegen wollen wir hier nicht ganz so viel forschen.

Why is recursion so slow?

Wenn wir unsere algorithmische Analyse z.B. für das Power Problem betrachten, wo wir ja die Iteration mit der Rekursion vergleichen, dann würde man naiv erwarten, dass die Rekursion etwa zweimal langsamer ist als die Iteration. In Wirklichkeit sieht der Unterschied eher wie ein Faktor Tausend aus. Warum ist das? Um der Sache auf die Spur zu kommen, müssen wir uns erkundigen was denn auf der Maschinesprach-Ebene alles passiert wenn eine Methode aufgerufen wird.

Fragen

1. Kommt es bei Permutationen auf die Reihenfolge an?

2. Geben Sie ein Beispiel für einen *Divide and Conquer* Algorithmus.

3. Betrachten Sie den folgenden Code:

```
boolean search(List<String> names, String key) {
    for (int i=0; i < names.size(); i++) {
        if (names[i] == key) return true;
    }
    return false;
}
```

   Wie ist sein Laufzeitverhalten im besten, im schlimmsten und im durchschnittliche Fall?

4. Hat das "Tower of Hanoi" -Problem exponentielles oder faktorielles Laufzeitverhalten? Welches ist schlimmer? (Hinweis: Stirling's Formel)

5. Algorithmen können präzise oder ungenau sein. Geben Sie ein Beispiel für einen ungenauen Algorithmus.

6. Welche Art von asymptotischem Laufzeitverhalten (d.h. Big-O-Notation) hat der folgende Algorithmus?

```
double celsiusToFahrenheit(double temp) {
    return temp*9.0/5.0 + 32;
}
```

7. Geben Sie einen Algorithmus zur Schätzung der Anzahl der Personen in einem Fußballstadion, der von Ordnung O(log (n)) ist.

8. Wann würden Sie einen Algorithmus als effizient bezeichnen?

9. Schätzen Sie das Laufzeitverhalten (große O-Notation) für die folgenden Algorithmen:
   - $f(n) = 24*n + 14$
   - $f(n) = 54333*n$
   - $f(n) = 3*n^2 + 4$

10. Betrachten Sie den folgenden Code. Nehmen Sie an, dass eine Anweisung ca. 1ms dauert. Geben Sie zunächst eine grobe Formel, um die Laufzeit in Bezug auf n zu abzuschätzen. Dann nehmen Sie an, dass n = 1000 ist und dass eine Anweisung ca. 1ms dauert. Wie lange dauert es bis die Methode fertig ist?

```
public void selectionSortFast(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
                int minIndex = i;
                for (int j = i + 1; j < arr.length; j++) {
                        if (arr[j] < arr[minIndex]) {
                                minIndex = j;
                        }
                }
        }
}
```

Referenzen

Die Referenzen für dieses Kapitel sind etwas speziell. Es lohnt sich aber auf jeden Fall einen genaueren Blick auf das Buch von Bruno R. Preiss [7] zu werfen.

[1] Stirling's approximation, https://en.wikipedia.org/wiki/Stirling's_approximation

[2] Dynamic programming, https://en.wikipedia.org/wiki/Dynamic_programming

[3] Wikibook Algorithms, R.Impagliazzo, Ma.Shonle, M.Wilson, M.Krischik, https://en.wikibooks.org/wiki/Algorithms/Dynamic_Programming

[4] Sine, https://en.wikipedia.org/wiki/Sine

[5] Nvidia, Cg 3.1 Toolkit Documentation, http://http.developer.nvidia.com/Cg/sin.html

[6] Fibonacci number, https://en.wikipedia.org/wiki/Fibonacci_number

[7] Data Structures and Algorithms, Bruno R. Preiss, https://www.brpreiss.com/books/opus4/html/page457.html

Sorting

Schon Aschenputtel wußte wie wichtig es ist schnell sortieren zu können: "die guten ins Töpfchen, die schlechten ins Kröpfchen" [7]. Und auch der damalige Präsidentschaftskandidat und spätere Präsident, Barack Obama, wusste bei einem Interview mit Google's Eric Schmidt [8], dass Bubble Sort kein besonders guter Sortieralgorithmus ist. Aber was ist denn ein guter Sortieralgorithmus? Darum geht es in diesem Kapitel: wir stellen die vier wichtigsten Sortieralgorithmen vor, und zeigen deren jeweilgen Stärken und Schwächen. Wir werden auch sehen wie man gut mischt, sozusagen das Gegenstück zum Sortieren. Und schließlich werden wir sehen welche Verbindung zwischen Suche und Sortierung besteht.

Shuffle

Bevor wir uns mit dem Sortieren beschäftigen können, müssen wir uns erst einmal kurz mit dem Mischen auseinandersetzen. Sonst haben wir ja nichts zum sortieren. Das richtige Mischen ist gar nicht so einfach und man muss sich da schon ein paar Gedanken machen. Gott sei Dank hat das schon jemand gemacht: die Herren Fisher und Yates [1]. Der sogenannte Fisher-Yates Algorithmus wird heutzutage am häufigsten verwendet. Sehen wir ihn uns mal an: wir wollen ein gegebenes Array von Ganzzahlen mischen:

```
private void shuffle(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n; i++) {
        int j = (int)(Math.random() * n);
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
}
```

Eigentlich ganz einfach: wir gehen durch das ganze Array, beginnend mit dem ersten Element, und tauschen eines nach dem anderen mit einem anderen zufällig ausgewählten Element aus.

Eine etwas schnellere Version des gleichen Algorithmus [2] erinnert sich, dass der erste Teil des Arrays bereits gemischt ist und daher die Zufallszahl eigentlich nur aus der hinteren Hälfte kommen muss:

```
private static void shuffleFast(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n; i++) {
        int j = i+ (int)(Math.random() * (n-i));
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
}
```

Diese zweite Version ist ein klein wenig schneller, aber beide Algorithmen führen das Shuffling in linearer Zeit durch, also O(n). Obwohl das Mischen wie eine triviale Übung erscheinen mag, kann überraschend viel dabei schief gehen, z.B. beim Generieren der Zufallszahlen [1].

Sorting

Kommen wir endlich zum Sortieren. Nehmen wir an wir haben ein Array von Ganzzahlen, das wir gerne sortiert hätten. Wie macht man das? Nun in Java ist das ganz einfach:

```
int[] arrOfInts = { 5, 55, 2, 7, 45, 3, 1, 8, 23, 12 };

Arrays.sort(arrOfInts);
```

Wir verwenden einfach die Methode *sort()* der Klasse Arrays, die macht das. Ganz einfach, Kapitel zu Ende.

Nun das hier wäre kein Buch über Algorithmen wenn wir nicht auch etwas über Sortieralgorthmen erfahren würden. Deswegen, was macht denn *Arrays.sort()* eigentlich?

Selection Sort

Sortieralgorithmen gibt es wie Sand am Meer, im Duzent sind sie billiger. Wir fangen aber mit dem einfachsten an, dem *Selection Sort*. Im Prinzip kennt jeder den Selection Sort der schon mal Karten gespielt hat: man nimmt seinen Stapel, sucht nach der kleinsten und setzt die an den Anfang. Dann sucht man nach der nächst kleinsten und setzt die daneben. Das macht man solange bis man den Stapel durch hat. Das übersetzen wir jetzt so, dass es auch der Computer kapiert:

1. suche nach der kleinsten Zahl und setze sie an den Anfang;
2. suche nach der nächst kleinsten und setze sie daneben;
3. mach das solange bis du alle Zahlen durch hast.

In Java sieht das Ganze dann so aus:

```
 1.  public void sort(int arr[])  {
 2.      for (int i = 0; i < arr.length - 1; i++) {
 3.          int min = i;
 4.          for (int j = i + 1; j < arr.length; j++) {
 5.              if (arr[j] < arr[min]) {
 6.                  min = j;
 7.              }
 8.          }
 9.          swap(arr, i, min);
10.      }
11.  }
```

Dabei tauscht die *swap()* Methode einfach zwei Elemente in dem Array:

```
private void swap(int arr[], int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
```

So schwer war das gar nicht. Die Frage die uns jetzt interessiert, wie lange dauert das denn? Betrachten wir die verschiedenen Teile:

1. die swap() Methode ist einfach, sie benötigt immer 3 Instruktionen;
2. Zeilen 5 und 6 sind ein Vergleich und evtl. eine Zuweisung, also im Schnitt 1.5 Instruktionen;
3. die innere for-Schleife, Zeilen 4 bis 8, besteht aus einem Vergleich und einem Inkrement, die bei jedem Schleifendurchlauf ausgeführt werden (2 Instruktionen), zusätzlich kommen noch die Zeilen 5 und 6 dazu. Insgesamt besteht also die innere Schleife aus 3.5 Instruktionen. Wie oft die Schleife durchlaufen wird hängt von der Größe des Arrays ab. Wenn man kurz darüber nachdenkt, dann ist das so im Schnitt n/2 mal, wobei *n* die Größe des Arrays ist. Das macht dann 3.5 * n/2 Instruktionen;
4. die äußere Schleife besteht wieder aus einem Vergleich und einem Inkrement (2 Instruktionen), dazu kommt die Zuweisung in Zeile 3, und natürlich kommt noch die innere Schleife dazu (3.5 * n/2 Instruktionen), dann noch der Swap (3 Instruktionen), macht dann insgesamt 6 + 3.5*n/2 Instruktionen pro Durchlauf;

Da die Schleife n-mal ausgeführt wird, kommt die Summe grob auf

```
# of instructions = n * ( 6 + 3.5*n/2 ) = 6*n + 1.75*n² ~ n²
```

Was wir hier also haben, ist ein Algorithmus mit quadratischer Laufzeit, $O(n^2)$, da der n^2 Term zum dominanten Term für große n wird. Quadratische Laufzeit ist schlecht. Eigentlich immer wenn man zwei for-Schleifen sieht deutet das auf $O(n^2)$ hin.

Insertion Sort

Kommen wir zur zweiten Sortiermethode, dem *Insertion Sort*. Der funktioniert so: man nimmt einfach die oberste Karte vom Stapel. Dann nimmt man die nächste Karte vom Stapel, und fügt die vor die erste Karte wenn sie kleiner ist, oder hinter die erste wenn sie größer ist. Dann kommt die nächste Karte vom Stapel, und die wird dann an die richtige Stelle "einsortiert". Wir übersetzen das wieder für den Computer:

1. nimm die erste Zahl vom Array, die ist sortiert;
2. dann nimm die zweite Zahl, wenn die kleiner als die erste ist, setze sie davor, ansonsten dahinter;
3. mach das mit den anderen Zahlen aus dem Array, eine nach der anderen, und füge sie in dem neuen Array jeweils an der richtigen Stelle ein.

Daraus wird dann der folgende Code:

```
 1.  public void sort(int arr[]) {
 2.      for (int i = 1; i < arr.length; i++) {
 3.          int cur = arr[i];
 4.          int j = i - 1;
 5.          while (j >= 0 && arr[j] > cur) {
 6.              arr[j + 1] = arr[j];
 7.              arr[j] = cur;
 8.              j--;
 9.          }
10.      }
11.  }
```

Wie schnell ist denn der Insertion Sort? Wir schauen uns die verschiedenen Teile des Codes an:

1. Zeile 6 ist ein Inkrement und eine Zuweisung, also 2 Instruktionen;
2. Zeilen 7 und 8 sind jeweils 1 Instruktion;
3. Zeile 5 besteht aus zwei Vergleichen und einer && Operation, macht 3 Instruktionen;
4. die innere while-Schleife, Zeilen 5 bis 9, wird im Schnitt n/2 mal augeführt, damit benötigt die innere Schleife 7 * n/2 Instruktionen;
5. Zeile 3 ist 1 Instruktion und Zeile 4 ist 2 Instruktionen;
6. Zeile 2 ist eine Instruktion und ein Vergleich, macht 2 Instruktionen.

Da die Schleife wieder n-mal ausgeführt wird, kommt das grob auf

```
# of instructions = n * ( 5 + 7*n/2 ) = 6*n + 3.5*n² ~ n²
```

Also auch der Insertion Sort hat quadratische Laufzeit, $O(n^2)$.

MergeSort

Bisher haben wir zwei Sortieralgorithmen gesehen, beide haben quadratische Laufzeit. Das bedeutet, wenn wir die Größe unseres Arrays verdoppeln, dann vervierfacht sich die Zeit die das dauert. Nun hat sich ein schlauer Mensch (John von Neumann) gedacht, wenn wir ein Array nehmen das nur halb so groß ist, dann würde es nur ein Viertel der Zeit dauern. Und genau das ist die Idee hinter dem *Merge Sort* Algorithmus:

- teile das Array in zwei Hälften;
- rekursiv, sortiere jede Hälfte;
- am Ende füge jeweils beide Hälften zusammen.

Man nennt den Merge Sort auch '**easy-split, hard-join**', denn das Aufspalten ist sehr einfach, die Arrays werden einfach halbiert. Die eigentliche Arbeit (also das Sortieren) erfolgt beim Zusammenfügen.

Schauen wir uns den Code an:

```
public void sort(int arr[]) {
    mergeSort(arr);
}
```

dabei ist die *mergeSort()* Methode rekursiv:

```
 1.  private void mergeSort(int[] arr) {
 2.      if (arr.length > 1) {
 3.          int middle = arr.length / 2;
 4.          int[] left = Arrays.copyOfRange(arr, 0, middle);
 5.          int[] right = Arrays.copyOfRange(arr, middle, arr.length);
 6.          mergeSort(left);
 7.          mergeSort(right);
 8.          merge(arr, left, right);
 9.      }
10.  }
```

Wie gesagt, die harte Arbeit steckt in der *merge()* Methode:

```
private void merge(int[] arr, int[] left, int[] right) {
    int pos = 0;
    int leftPos = 0;
    int rightPos = 0;

    // main merge loop
    while (leftPos < left.length && rightPos < right.length) {
        if (left[leftPos] < right[rightPos]) {
            arr[pos] = left[leftPos];
            leftPos++;
        } else {
            arr[pos] = right[rightPos];
            rightPos++;
        }
        pos++;
    }

    // copy rest of left half, if needed
    while (leftPos < left.length) {
        arr[pos] = left[leftPos];
        leftPos++;
        pos++;
    }

    // copy rest of right half, if needed
    while (rightPos < right.length) {
        arr[pos] = right[rightPos];
        rightPos++;
        pos++;
    }
}
```

Und wie lange dauert das? Das ist ein bisschen knifflig, aber fangen wir langsam an. Die Zeit, die in der Methode *merge()* verbracht wird, ist proportional zur Größe des Arrays. Um das zu sehen, gehen wir mal davon aus, dass die linken und rechten Arrays so sind, dass nach dem "main merge loop" beide leer sind. Dann sind die Zuweisungen in den ersten drei Zeilen 3 Instruktionen. Der innere Teil der while-Schleife ist ein Vergleich, eine Zuweisung und zwei Inkrements, d.h. 4 Befehle. Der Vergleich der while-Schleife selbst besteht aus zwei Vergleichen und einem &&, also 3 Instruktionen, also zusammen beinhaltet die while-Schleife 7 * n Instruktionen. Das bedeutet,

```
# of instructions for merge() = 3 + 7*n
```

Was bleibt ist die *mergeSort()* Methode: Zeile 2 ist ein Vergleich, 1 Instruktion. Zeile 3 ist eine Division und eine Zuordnung, 2 Instruktionen. Zeile 4 und 5 sind interessant: In Java müssen wir eine Kopie machen, aber in C oder C++ wäre es mit Zeiger Arithmetik einfach, ein Array zu halbieren. Da wir uns nur für den Algorithmus interessieren und nicht für Probleme die mit Java oder irgendeiner anderen Sprache zu tun haben, nehmen wir an, dass jede dieser Zeilen nur eine Instruktion ist.

Was bleibt ist der rekursive Aufruf. Das ist nicht ganz einfach, aber im letzten Kapitel hatten wir mit der *powerDC()* Methode ein ganz ähnliches Problem. Dort haben wir, um ein Gefühl für das Problem zu bekommen, einfach mal geschaut wie lange es dauert für verschieden große Arrays. Beginnen wir mit einem Array, das nur ein Element hat, also n=1:

- **n=1:** es gibt keinen rekursiven Aufruf, nur einen Vergleich in Zeile 2, also insgesamt 1 Instruktion;
- **n=2:** Zeilen 2 bis 5 entsprechen 5 Instruktionen, Zeilen 6 und 7 benötigen je 1 Instruktion, und der merge() in Zeile 8 entspricht 3+7*n = 17 Instruktionen, zusammen also 5+2*1+17 = 24 Instruktionen;
- **n=4:** hier wird n=2 zweimal rekursiv aufgerufen, macht 2*24 Instruktionen, der merge in Zeile 8 entspricht 3+7*n= 31 Instruktionen, was insgesamt zu 5+2*24+31 = 84 Instruktionen führt;
- **n=8:** hier wird n=4 zweimal rekursiv aufgerufen, macht 2*84 Instruktionen, der merge in Zeile 8 entspricht 3+7*n= 59 Instruktionen, was insgesamt 5+2*84+59 = 232 Instruktionen bedeutet;
- **n=16:** hier wird n=8 zweimal rekursiv aufgerufen, macht 2*232 Instruktionen, der merge in Zeile 8 entspricht 3+7*n= 115 Instruktionen, was insgesamt 5+2*232+115 = 584 Instruktionen bedeutet.

Wir könnten jetzt zwar so weitermachen, aber was wir haben genügt schon. Wir tragen unsere Ergebnisse mal in eine Tabelle ein, und vergleichen sie mit *n*, mit *n*log(n)* und mit n^2:

log(n)	n	n * log(n)	n^2	10* n * log(n)	merge()	4* n^2
1	2	2	4	20	24	16
2	4	8	16	80	84	64
3	8	24	64	240	232	256
4	16	64	256	640	584	1024
5	32	160	1024	1600	1400	4096

Das ist jetzt kein exakter mathematischer Beweis, aber wenn man sich die Tabelle ansieht, dann ist ziemlich klar, dass sich merge() am ehesten wie *n*log(n)* verhält und nicht wie n^2. Man kann tatsächlich beweisen, dass das der Fall ist. Dies ist das erste Mal, dass wir auf einen Algorithmus stoßen, der ein *linearithmisches* Laufzeitverhalten hat, d.h. O(n*log(n)). Und das ist eine gute Sache. Merge Sort gehört auch zur Klasse der *divide and conquer* Algorithmen.

QuickSort

So wie Merge Sort ist auch *Quick Sort* ein *divide and conquer* Algorithmus. Die Grundidee ist folgende:

- teile das Array in zwei Hälften, eine mit den niedrigeren Zahlen und eine mit den höheren;
- sortiere jede Hälfte, rekursiv;
- am Ende füge die beiden Hälften einfach aneinander.

Man nennt diese Methode auch '**hard-split, easy-join**'. Hier ist das Aufspalten der schwierige Teil, dafür ist das Zusammenfügen einfach. In Bezug auf den Code sieht das so aus:

```
public void sort(int arr[]) {
    quickSort(arr);
}
```

wobei die *quickSort()* Methode rekursiv ist und sich selbst aufruft:

```
private void quickSort(int[] arr) {
    if (arr.length > 1) {
        int pivot = arr[arr.length / 2];
        int[] low = getLowerHalf(arr, pivot);
        int[] high = getUpperHalf(arr, pivot);
        quickSort(low);
        quickSort(high);
        join(arr, low, high);
    }

}
```

Wir wollen uns hier nicht mit den Details langweilen, aber man kann auch zeigen, dass Quick Sort linearithmisches Laufzeitverhalten hat. Der einfache Quick Sort hat ein paar kleine Macken, aber die kann man alle beheben.

So, nun zurück zu unserer ursprünglichen Frage: welchen Algorithmus verwendet denn Arrays.sort()? Wenn wir in der Java Dokumentation der Klasse Arrays nachschlagen, finden wir, dass ein modifizierter und optimierter Quick Sort Algorithmus verwendet wird [3].

Which one is the best?

Normalerweise ist ein linearithmischer Algorithmus immer besser als ein quadratischer. Aber es gibt Ausnahmen, z.B.:

- Insertion Sort ist sehr schnell, wenn die Ausgangsdaten bereits fast sortiert sind. Interessanterweise passiert das relativ häufig in der realen Welt.
- Selection Sort minimiert die Anzahl der Swaps. Also für den Fall, dass ein Swap eine sehr teure Operation ist (z.B. schwere Möbel verschieben), dann kann der Selection Sort die bessere Wahl sein.

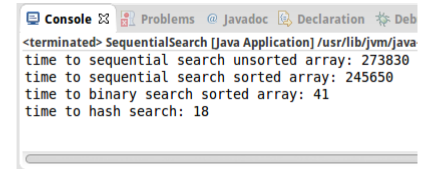
Es gibt auch Fälle, in denen z.B. Quick Sort nicht so gut ist. Bleibt die Frage, geht es vielleicht noch schneller als linearithmisch? Die Antwort liefert die Theorie: man kann zeigen, dass es keinen Sortieralgorithmus gibt der schneller als O(n * log(n)) ist. Das ist o.k. Wie bereits angedeutet, gibt es noch ein paar mehr Sortieralgorithmen, falls Interesse besteht kann man sich die im Internet ansehen [4], die werden auch alle mit einer schönen Animation verglichen, wirklich hübsch.

Review

Wir haben kurz gesehen wie man richtig mischt, danach haben wir uns aber hauptsächlich mit dem Sortieren beschäftigt. Wir haben vier Sortieralgorithmen näher kennen gelernt, and the winner is: QuickSort.

Projekte

In diesem Kapitel gibt es nur wenige Projekte, was nicht heißen soll, dass sie unwichtig sind. Suche ist nämlich ein sehr wichtiges Thema.



Searching

Suchen und Sortieren gehen Hand in Hand. Um das zu zeigen schauen wir uns mehrere verschiedene Möglichkeiten an zu suchen. Als Beispiel verwenden wir folgendes Array von Ganzzahlen:

```
int[] arr = { 44, 88, 17, 32, 97, 65, 28, 82, 29,
76, 54, 80 };
```

Darin wollen wir nach der Zahl 17 suchen, die an Position 2 ist, und nach der Zahl 42, die gar nicht in dem Array ist.

Sequential Search

Als erstes versuchen wir es mal mit der *Sequential Search*: das ist ein Brute-Force Algorithmus, der einfach alle Einträge des Arrays durchläuft, bis die Zahl die wir suchen, gefunden wird:

```
private int sequentialSearch(int key, int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;

}
```

Manchmal wird die Zahl die wir suchen, eher am Anfang des Arrays sein, manchmal eher am Ende. Im Durchschnitt müssen wir also ca. durch die Hälfte des Arrays gehen, wenn die gesuchte Zahl im Array ist, oder wenn sie gar nicht im Array ist, durch das ganze Array. Wir erwarten also ein Laufzeitverhalten, das linear zur Größe des Arrays ist, d.h. O(n).

Sequential Search2

In einem zweiten Versuch nehmen wir an, dass unser Array sortiert ist:

```
int[] arr = { 17, 28, 29, 32, 44, 54, 65, 76, 80, 82, 88, 97 };
```

Wenn wir aber kurz nachdenken, ändert das gar nichts. Das Laufzeitverhalten bleibt bei O(n).

Binary Search

Aber wir geben nicht so schnell auf: wie wäre es mit einem *divide and conquer* Ansatz? Angenommen, wir suchen die Zahl 17 und das Array ist sortiert. Dann folgt der *Binary Search* Algorithmus diesen Schritten:

1. nimm die Zahl in der Mitte des Arrays (das ist die 54);
2. vergleiche sie mit der gesuchten 17;
3. sind die beiden gleich, sind wir fertig, wir haben die Zahl gefunden;
4. ist 17 kleiner als die Mitte, dann wissen wir, dass 17 unmöglich in der rechten Hälfte des Arrays sein kann, weil es ja sortiert ist, können dort nur Zahlen größer als 54 sein. Also müssen wir für unsere weitere Suche nur in der linken Hälfte suchen. Unser Problem hat sich gerade um die Hälfte reduziert;

5. jetzt gehen wir wieder zurück zum ersten Schritt, allerdings nur mit der linken Hälfte. Das machen wir so lange bis wir entweder die Zahl gefunden haben, oder bis das Array nur noch ein Element hat.

In Java sieht der Binary Search Algorithmus wie folgt aus:

```
private static int binarySearch(int key, int[] arr, int start, int stop)
{
    // base case
    if (start > stop)
        return -1;

    // recursive case
    int mid = (start + stop) / 2;
    if (key == arr[mid]) {
        return mid;
    } else if (key < arr[mid]) {
        return binarySearch(key, arr, start, mid - 1);
    } else {
        return binarySearch(key, arr, mid + 1, stop);
    }

}
```

Das Laufzeitverhalten des binären Suchalgorithmus ist O(log(n)), also deutlich besser als die sequentielle Suche. Das ist einer der Gründe, warum Sortieren so wichtig ist, weil es die Suchzeiten drastisch reduzieren hilft.

Hash Search

Ist die binäre Suche wirklich die schnellste? Und ist die Sortiererei wirklich nötig? Dauert ja auch. Zweimal Nein. Das Sortieren ist nicht unbedingt nötig, und binäre Suche ist auch nicht die schnellste. *Hash-based Search* ist die schnellste. Wie funktioniert sie? Wir brauchen zwei Schritte: im ersten speichern wir unser Array in einem HashSet:

```
HashSet<Integer> hs = new HashSet(Arrays.asList(arr));
```

und im zweiten verwenden wir einfach die *contains()* Methode des HashSets für die Suche:

```
boolean b = hs.contains(key);
```

Die Suche selbst hat konstantes Laufzeitverhalten, also O(1). Das ist also noch schneller als O(log(n)). Ausserdem muss unser Array nicht sortiert werden, also wir sparen uns die O(n log(n)) Strafe für das Sortieren. Aber wir müssen unser Array in das HashSet einfügen, und das kostet uns O(n). Trotzdem am Ende noch deutlich schneller!

```
sequential search unsorted array: 273830 ms
sequential search sorted array:   245650 ms
binary search sorted array:           41 ms
hash search:                          18 ms
```

Obwohl Hash-based Search doppelt so schnell wie Binäre Suche ist, macht es fast keinen Unterschied wenn man beide mit sequentieller Suche vergleicht.

Binary Search Tree

Es gibt noch eine Suchmethode, die wir in diesem Zusammenhang erwähnen wollen: der binäre Suchbaum. Wir werden sie im Kapitel über Bäume kennenlernen. Sie hat auch logarithmische Suchzeiten, d.h. O(log (n)), und hat auch noch einige andere Vorteile. Dennoch, wenn es um rohe Suchkraft geht: hash is the best!

TreeMap

In Kapitel drei haben wir bereits in zwei Projekten Bekanntschaft mit der TreeMap gemacht: in Languages und in BuildIndex. Das ist eine ganz normale Map bei der allerdings die Schlüssel automatisch sortiert sind. Welchen Sortieralgorithmus verwendet denn die TreeMap? Interessanterweise keinen der hier behandelten. Anstelle verwendet es einen *Red-Black Tree* als interne Datenstruktur, der für die Sortierung sorgt. Wir werden mehr dazu im nächsten Kapitel erfahren.

Research

Ein Thema das wir in diesem Buch komplett ausgelassen haben ist der Heap, auch eine interessante Datenstruktur.

HeapSort

Basierend auf dem Heap gibt es auch ein Sortierverfahren, den HeapSort. Wir sollten vielleicht erst mal nachlesen was denn überhaupt ein Heap ist und danach könnten wir uns mal den HeapSort näher ansehen.

Fragen

1. Bitte erläutern Sie, wie der SelektionSort und der InsertionSort funktionieren.

2. Wie gut ist QuickSort bei der Sortierung bereits sortierter Daten? Wie gut ist InsertionSort beim Sortieren bereits sortierter Daten?

3. Sie haben eine Gruppe von Studenten, die zufällig im Wohnheim auf Zimmer verteilt wurden. Da die Zimmerhöhe aber von hinten nach vorne im Wohnheim abnimmt, wollen Sie die Räume neu zuordnen, so dass die kleineren Studenten in den hinteren Räumen sind, während die größeren in den vorderen. Jeder Student hat eine Menge persönliches Zeug zu schleppen, deswegen sollte die Anzahl der Umzüge minimiert werden. Würden Sie eher einen InsertionSort oder einen SelektionSort Algorithmus verwenden? Warum?

.

Referenzen

Zwei hervorragende Bücher wenn es um Suche geht sind das Buch von Roberts und Zelenski [5], sowie das von Sedgewick and Wayne [2]. Das Buch von Goodrich und Tamassia [6] steigt etwas tiefer in die Materie ein, ist auch sehr zu empfehlen.

[1] Fisher–Yates shuffle, https://en.wikipedia.org/wiki/Fisher–Yates_shuffle#The_modern_algorithm

[2] Introduction to Programming in Java, Robert Sedgewick and Kevin Wayne

[3] Arrays (Java Platform SE 7) - Oracle, https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

[4] Sorting Algorithms Animations, http://www.sorting-algorithms.com/

[5] Programming Abstractions in C++, Eric S. Roberts and Julie Zelenski

[6] Data Structures and Algorithms in Java, M.T. Goodrich and R. Tamassia

[7] Aschenputtel, https://de.wikipedia.org/wiki/Aschenputtel_(1989)

[8] Barack Obama | Candidates at Google, https://www.youtube.com/watch?v=m4yVlPqeZwo, (about 23 minutes into the talk)

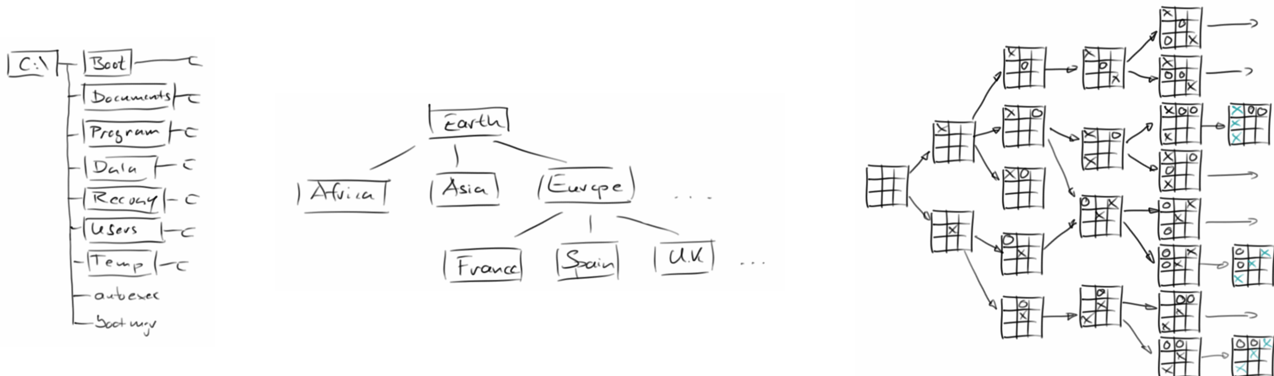
Trees

Ich liebe Bäume. In meinem Garten habe ich ein paar Kirschbäume, einen Maulbeerbaum, einen Mispelbaum, natürlich ein paar Äpfelbäume, einen kleinen Pfirsich-, und einen Feigenbaum, der jetzt schon drei Winter überlebt hat. Aber da das hier kein Buch für Hobbygärtner ist sondern für angehende Algorithmiker, geht es in diesem Kapitel um die Datenstruktur Baum, auf Englisch *Tree*.

Bei Trees handelt es sich um eine hierarchische Datenstruktur, die wenn man sie ausdruckt etwas an Bäume erinnert, daher der Name. So wie es bei richtigen Bäumen ganz viele Arten von Bäumen gibt, ist das auch bei der Datenstruktur, da gibt es BinaryTrees, QuadTrees, Splay Trees, Red-Black Trees, usw. Und je nachdem welches Obst man essen möchte muss man auch die richtige Baumart auswählen. Darum geht es in diesem Kapitel, um das Hegen und Pflegen von Bäumen.

Examples

Beginnen wir mit ein paar Beispielen: die Dateien und Verzeichnisse auf unserem Computer sind ein Baum. Wenn wir unsere Erde in Kontinente, Staaten, Bundesländer, usw. unterteilen, dann ist das auch ein Baum. Oder z.B. die Züge in einem Spiel, wie z.B. Tic-Tac-Toe, bilden auch eine Baumstruktur:

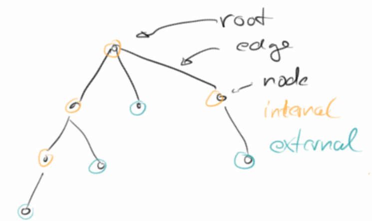


Es gibt noch viele weitere Beispiele:

- Familienbäume, also gemeint sind Stammbäume.
- Ein Buch bestehend aus Titel, Inhaltsverzeichnis, Kapiteln, Unterkapiteln und Paragraphen.
- Entscheidungsbäume, basierend auf mehreren konsekutiven Ja/Nein Fragen kann man eine Entscheidung oder Diagnose treffen.
- Mathematische arithmetische Ausdrücke haben eine Baumstruktur.
- Und viele Spiele lassen sich in eine Baumstruktur abbilden.

Allerdings nicht alles was wie ein Baum aussieht ist wirklich ein Baum, deswegen wollen wir erst einmal definieren, was wir unter einem Baum verstehen.

Terminology

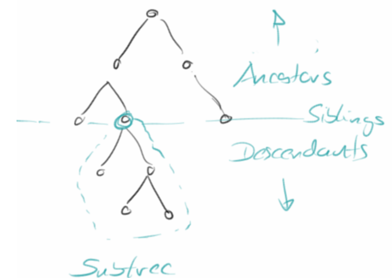


Ein Baum ist wie eine Eltern-Kind Beziehung. Dabei kann das Elternteil mehrere Kinder haben, aber ein Kind kann nie mehr als ein Elternteil haben. Das mag etwas ungewöhnlich erscheinen, aber so ist die Definition.

Ein Baum besteht aus Knoten, *Nodes* genannt, und Verbindungen, die man *Edges* nennt. Des weiteren verwendet man folgende Begriffe:

- Root: die Wurzel ist der Knoten der kein Elternteil hat. In jedem Baum kann es nur ein Wurzel geben.
- Internal Node: das sind Knoten die selbst Kinder haben.
- External Node (Leaf): das sind Knoten die keine Kinder haben.

In Anlehnung an Familienbäume, werden dann auch die folgenden Bezeichnungen verwendet:



- Ancestors: sind die Vorfahren.
- Descendants: sind die Nachfahren, also Kinder und Kinderskinder.
- Siblings: nennt man die Geschwister eines Knotens.
- Subtree: ein Unterbaum oder Teilbaum, ist ein Teil eines größeren Baumes, der selbst wieder ein Baum ist.
- Depth: bezieht sich auf einen Knoten und ist die Distanz dieses Knotens zum Wurzelknoten. Der Wurzelknoten hat Depth von 0.
- Height: ist das Gegenstück zur Depth, und bezieht sich auf die externen Knoten. Externe Knoten haben eine Height von 0. Die Height des Baumes entspricht dann der Height des Wurzelknotens.

Schließlich unterscheidet man noch zwischen geordneten und ungeordneten Bäumen: das bezieht sich auf die Kinder: gibt es da eine Reihenfolge (wie z.B. deren Alter) oder gibt es da keine. Wir werden uns nur mit geordneten Bäumen beschäftigen.

Nodes

Die Datenstruktur *Node* besteht aus folgenden Teilen: es gibt eine Referenz auf das Elternteil, es gibt Referenzen auf mögliche Kinder und es gibt eine Referenz auf das *Element*, das sind die Daten die im Knoten gespeichert werden. Die Datenstruktur *Node* unterstützt die folgenden, lesenden Methoden:

- **isRoot():** gibt an ob es sich bei dem Knoten um den Wurzelknoten handelt;
- **isInternal():** wenn ein Knoten Kinder hat, dann ist es ein interner Knoten;
- **isExternal():** wenn ein Knoten keine Kinder hat, dann ist es ein externer Knoten;
- **getElement():** gibt das Element zurück, das in diesem Knoten gespeichert ist;
- **getParent():** gibt den übergeordneten Knoten dieses Knotens zurück;
- **getChildren():** gibt die untergeordneten Knoten dieses Knotens zurück.

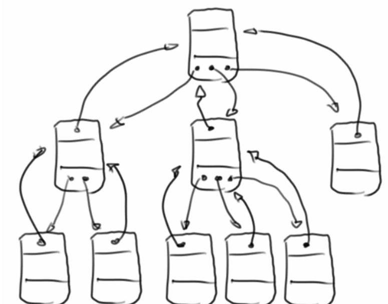
Darüber hinaus kann ein Knoten auch Schreibmethoden haben:

- **setElement(E element):** setzt das Element dieses Knotens auf einen neuen Wert;
- **addChild(Node<E> node):** fügt diesem Knoten ein Kind hinzu;
- **removeChild(Node<E> node):** entfernt das gegebene Kind von diesem Knotens, wenn dieses Kind wiederum Kinder hat, werden diese auch mit entfernt!

Mit diesen Methoden können wir Strukturen von Knoten zusammen bauen, in dem wir Information in ihnen speichern und sie dann in einer Struktur verbinden. Als Beispiel wollen wir den Baum der ersten biblischen Familie aufbauen:

```
Node<String> adam = new Node<String>("Adam");
Node<String> abel = new Node<String>("Abel");
adam.addChild(abel);
Node<String> cain = new Node<String>("Cain");
adam.addChild(cain);

System.out.println(adam.isRoot());
System.out.println(adam.getChildren());


adam.removeChild(abel);  // Abel was killed by his brother
```

Ordered Trees



Wenn wir anfangen Knoten zu verbinden, bekommen wir Bäume. Bäume bestehen aus Knoten. Obwohl wir im Prinzip schon mit einem Haufen von Knoten ganz gut arbeiten könnten, bietet die übergreifende *Tree* Datenstruktur einige zusätzliche, nützliche Funktionen:

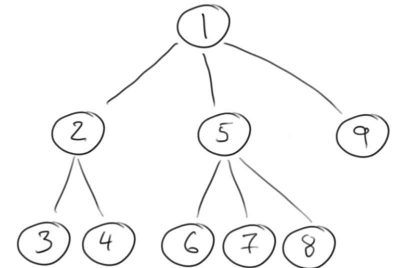
- **root():** gibt den Wurzelknoten des Baums zurück;
- **size():** gibt die Größe des Baumes, d.h. die Anzahl der Knoten im gesamten Baumes;
- **height():** gibt die Höhe eines Baumes;
- **elements():** gibt alle Knoten des Baumes als Collection zurück;
- **preOrder():** eine Art, einen Baum zu durchqueren;
- **postOrder():** eine andere Möglichkeit, einen Baum zu durchlaufen;
- **levelOrder():** noch eine andere Art, durch einen Baum zu iterieren.

Diese Methoden sind ganz praktisch wie wir in Kürze sehen werden. Verwendet wird die Datenstruktur wie folgt:

```
Tree<String> humans = new Tree<String>(adam);
System.out.println(humans.size());
System.out.println(humans.elements());

humans.preOrder();
```

Besonders die Traversal, also die Iterationsmethoden werden für uns sehr wichtig werden. Deswegen betrachten wir diese jetzt ein wenig näher.

Pre-Order Traversal



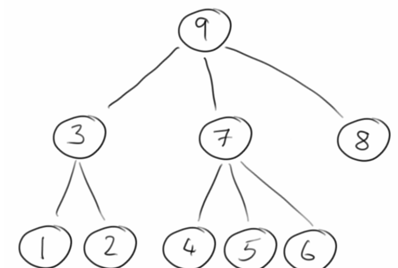
Wenn wir durch alle Elemente eines Baumes iterieren wollen, haben wir mehrere Möglichkeiten, eine davon ist das *Pre-Order* Traversal. Bei dieser Iteration wird der Knoten selbst vor seinen Kindern besucht.

```
void preOrder(node) {
    visit(node)
    for (child of node) {
        preOrder(child)
    }

}
```

Mögliche Anwendungen für dieser Art von Traversal sind:

- das Drucken eines strukturierten Dokuments (wie z.B. eines Buchs);
- einen Baum als String auszugeben (z.B. new OrderedTreeParser().createStringFromTree());
- oder einen Baum zu malen (FibonacciTree).

Post-Order Traversal



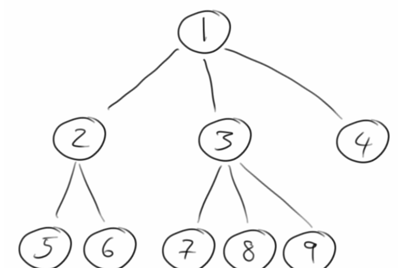
Wenden wir uns der nächsten Form der Iteration zu, dem *Post-Order* Traversal. In dieser wird der Knoten erst nach seinen Kinder besucht.

```
void postOrder(node) {
    for (child of node) {
        postOrder(child)
    }
    visit(p)

}
```

Mögliche Anwendungen dieser Art von Durchquerung sind:

- das Berechnen des Speicherplatzes von Dateien in Verzeichnissen und deren Unterverzeichnissen;
- und das Auswerten von arithmetischen Ausdrücken.

Level-Order Traversal



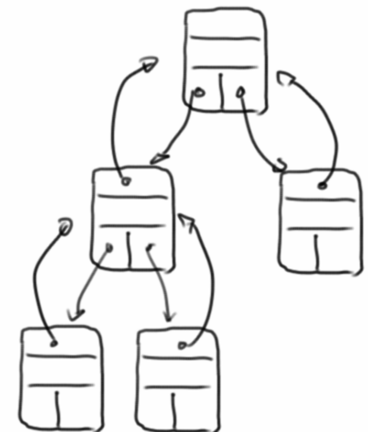
Als letzte wichtige Form der Iteration durch geordente Bäume betrachten wir kurz das *Level-Order* Traversal. Hier wird zu erst der Wurzelknoten besucht, danach all seine Kinder, dann kommen die Enkelkinder, usw..

```
void levelOrder(node, level) {
    if (node == null) {
        return;
    } else if (level == 1) {
        visit(node);
    } else if (level > 1) {
        for (child of node) {
            levelOrder(child, level - 1);
        }
    }
}
```

Diese Art von Traversal wird verwendet

- um z.B. durch einen Baum hierarchisch von oben nach unten durch zu iterieren.

Binary Trees



Eine besondere Form von Bäumen sind binäre Bäume:

- Jeder Knoten hat höchstens zwei Kinder und
- die Kinder haben eine Ordnung, d.h. es gibt ein linkes und ein rechtes Kind.

Wenn ein binärer Baum keine oder genau zwei Kinder hat, dann nennt man ihn auch Proper Binary Tree.

Ein binärer Baum, *BinaryTree*, besteht aus binären Knoten, *BinaryNode*. Binäre Knoten haben die gleichen Operationen wie normale Knoten, nur zusätzlich haben sie noch folgende Methoden:

- **hasLeft():** true, wenn dieser Knoten ein Kind auf der linken Seite hat;
- **hasRight():** true, wenn dieser Knoten ein Kind auf der rechten Seite hat;
- **getLeft():** gibt das Kind auf der linken Seite dieses Knotens zurück;
- **getRight():** gibt das Kind auf der rechten Seite dieses Knotens zurück;
- **setLeft(BinaryNode<E> node):** fügt ein Kind auf der linken Seite ein;
- **setRight(BinaryNode<E> node):** fügt ein Kind auf der rechten Seite ein;
- **removeLeft():** löscht das Kind auf der linken Seite;
- **removeRight():** löscht das Kind auf der rechten Seite.

Zusätzlich zu den Methoden, die für generische Bäume definiert sind, haben binäre Bäume die folgende Operation,

- **inOrder():** eine vierte Möglichkeit, einen Baum zu durchqueren, funktioniert aber nur für binäre Bäume.

Um zu sehen, wie wir mit binären Bäumen arbeiten, können wir wieder unsere erste Familie betrachten:

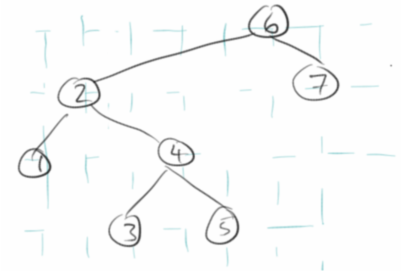
```
BinaryNode<String> adam = new BinaryNode<String>("Adam");
BinaryNode<String> cain = new BinaryNode<String>("Cain");
cain.setLeft(new BinaryNode<String>("Enoch"));
adam.setLeft(cain);
BinaryNode<String> abel = new BinaryNode<String>("Abel");
adam.setRight(abel);

BinaryTree<String> humans = new BinaryTree<String>(adam);

System.out.println(humans);
System.out.println(humans.size());
System.out.println(humans.height());

humans.inOrder();
```

Wie wir sehen werden, haben binäre Bäume viele Anwendungen:

- Arithmetische Ausdrücke,
- Entscheidungsprozesse und
- Suchen.

Binäre Bäume sind so wichtig, denn sie haben viele schöne Eigenschaften. Darüber hinaus gibt es einen Satz der besagt, dass alle Bäume in binäre Bäume umgewandelt werden können [16]. Das ist ganz praktisch.

In-Order Traversal



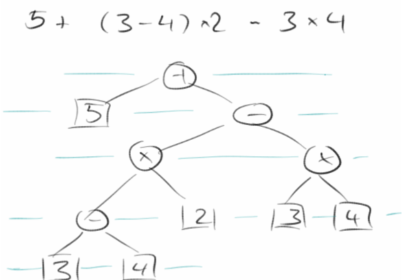
Das *In-Order* Traversal funktioniert nur mit binären Bäumen. Bei ihm wird zuerst die linke Seite besucht, dann wird der Knoten selbst besucht, und danach die rechte Seite. D.h. ein Knoten wird nach seinem linken Teilbaum, aber vor seinem rechten Teilbaum besucht:

```
void inOrder(p) {
    if hasLeft(p) {
        inOrder( left(p) );
    }
    visit(p);
    if hasRight(p) {
        inOrder( right(p) );
    }

}
```

Mögliche Anwendungen dieser Art von Durchquerung sind:

- Das Drucken arithmetischer Ausdrücke und andere Binärbäume in einer hübschen Form;
- Wenn man mit diesem Traversal binäre Suchbäume durchläuft, werden die Elemente in sortierter Reihenfolge ausgegeben (BinarySearchTree.main()).

Arithmetic Expression Tree



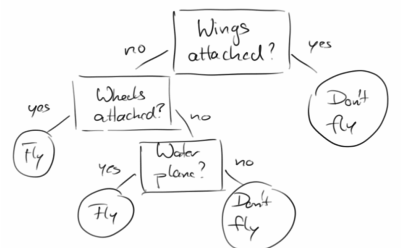
Eine wichtige Anwendung von binären Bäumen ist der arithmetische Ausdruck. Hier repräsentieren interne Knoten Operatoren und externe Knoten entsprechen den Operanden. Zum Beispiel, aus dem Ausdruck

```
5 + ( 3 - 4 ) * 2 - 3 * 4
```

wird der binäre Baum rechts. Für arithmetische Ausdrucksbäume haben zwei Arten von Traversals eine besondere Bedeutung:

- Post-Traversal kann verwendet werden, um arithmetische Ausdrücke zu berechnen;
- In-Order-Traversal kann verwendet werden, um arithmetische Ausdrücke schön darzustellen oder zu drucken.

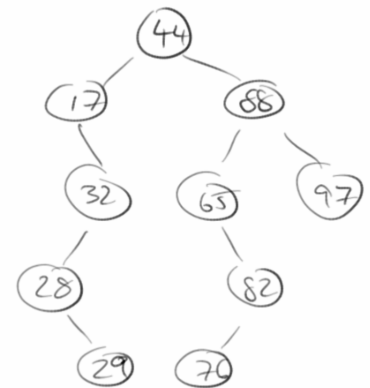
Decision Trees



Eine weitere schöne Anwendung eines binären Baumes ist der Entscheidungsbaum [15]. Bei einem Entscheidungsbaum sind interne Knoten Fragen. Bei einer Ja-Antwort folgt man dann der Frage auf der linken Seite, bei einer Nein-Antwort folgt man der Frage auf der rechten Seite. Externe Knoten entsprechen dann den Entscheidungen die zu treffen sind. Beispiele beinhalten:

- Checkliste für ein Flugzeug vor dem Start;
- Was man essen soll;
- Welchen Sortieralgorithmus man verwenden soll;
- Welche Programmiersprache sich für ein bestimmtes Problem am besten eignet.

Binary Search Trees



Eine sehr prominenten Anwendung von Bäumen ist die der Suche. Der *Binary Search Tree* ist dabei ein Beispiel das einen binären Baum als Datenstruktur verwendet. Dabei werden:

- interne Knoten verwendet um Einträge zu speichern;
- neue Einträge, je nachdem ob sie größer oder kleiner sind, in den rechten oder linken Teilbaum eingefügt.

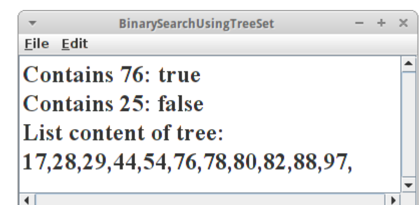
Bei dieser Baumart werden die "Blätter", also die externen Knoten, gar nicht genutzt. Betrachten wir folgendes Beispiel:

- **Create:** Beginnend mit einem leeren Suchbaum, fügen wir nacheinander die folgenden Zahlen ein: 44, 88, 17, 32, 97, 65, 28, 82, 29, 76, 54, 80. Es entsteht der Baum rechts.
- **Find:** Wir versuchen nach den Einträgen 76 und 25 zu suchen indem wir den Baum von oben nach unten durchgehen. Dabei vergleichen wir unsere Suchzahl, z.B. die 76 jeweils mit dem momentanen Knoten, und folgen der linken Seite wenn unsere Suchzahl größer ist und der rechten Seite wenn sie kleiner ist.
- **Insert:** Als nächstes fügen wir die Zahl 78 in den Baum ein.
- **Remove:** Das Entfernen von Elemente ist die schwierigste Operation. Versuchen wir die Zahlen 32 und 65 zu entfernen. Erst müssen wir die Zahl die wir suchen im Baum finden. Wenn die Suchzahl gar nicht im Baum ist dann ist das einfach. Auch wenn die Suchzahl ganz unten ist, dann löschen wir einfach diesen Knoten aus unserem Baum. Wenn unsere Suchzahl allerdings irgendwo mitten im Baum ist, und vielleicht auch linke und rechte Teilbäume unter sich hat, dann wird das etwas schwieriger. Die einfache Lösung ist einfach diese Teilbäume zu nehmen und ganz neu in den Baum einzufügen. Das ist allerdings nicht besonders schnell.

Was die Laufzeiten angeht kann man zeigen, dass das Suchen, Einfügen und Entfernen proportional zur Höhe des Baumes dauert, für einen ausgewogenen (well-balanced) Baum bedeutet dies O(log n).

Es müsste eigentlich klar sein, dass eine der effektivsten Lösungen für das NumberGuessGame nichts anderes als ein binärer Suchbaum ist.

TreeSet



Wir können zwar unsere eigene Klasse schreiben um die Binäre Suche umzusetzen (was wir später auch tun werden), aber warum das Rad neu erfinden, wenn es schon jemand anderes erfunden hat? In Java gibt es eine Klasse namens *TreeSet*. Und die macht genau was wir eigentlich wollen. Hier eine Umsetzung der Aufgabe von gerade eben:

```
int[] nrs = { 44, 88, 17, 32, 97, 65, 28, 82, 29, 76, 54, 80 };

// create tree and insert
TreeSet<Integer> searchTree = new TreeSet<Integer>();
for (int i = 0; i < nrs.length; i++) {
    searchTree.add(nrs[i]);
}
```

```
// find the entries: 76 and 25
println("Contains 76: " + searchTree.contains(76));
println("Contains 25: " + searchTree.contains(25));

// insert the number 78
searchTree.add(78);

// remove numbers 32 and 65
searchTree.remove(32);
searchTree.remove(65);

// print whole tree
println("List content of tree: ");
for (Integer nr : searchTree) {
    print(nr + ",");

}
```

Der Vorteil des TreeSets ist, dass seine Einträge sortiert sind. Dabei passiert das Sortieren beim Einfügen neuer Elemente. Wenn wir nach einem Eintrag suchen, dann dauert das O(log n), ist also langsamer als beim HashSet, dafür ist es aber sortiert. Intern verwendet das TreeSet die *Red-BlackTrees* als Datenstruktur. Das Gleiche gilt auch für die TreeMap.

Other Trees

Wir haben uns bisher nur mit Trees beschäftigt die auf einer Verlinkung der Knoten beruhen. In den Projekten werden wir noch ein Beispiel sehen, das auf Arrays beruht. Es gibt aber noch ganz viele andere Implementierungen für die Tree Datenstruktur, z.B. sind das:

- QuadTrees
- AVL Trees
- Splay Trees
- (2,4) Trees
- Red-Black Trees
- B-Trees

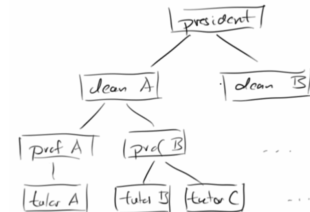
Ähnlich wie bei den Sortierverfahren hat jede so ihre Stärken und Schwächen. Aber was der QuickSort für die Sortieralgorithmen ist, das ist der Red-Black Tree für Bäume.

Review

In diesem Kapitel haben wir uns mit Bäumen angefreundet. Dabei haben wir geordnete Bäume und binäre Bäume gesehen und wir haben von den verschiedenen Traversal-Methoden gehört. Und der Binary Search Tree ist uns kurz über den Weg gelaufen.

Projekte

Es gibt überraschend viele Anwendungen für Bäume. Sie lassen sich sehr gut dazu verwenden um hierarchische Strukturen abzubilden. Man kann sie für Entscheidungsbäume nutzen. Häufig sind sie das Resultat von Parsern, man kann mit ihnen suchen und sortieren, und sowohl für die Auswertung von arithmetischen Ausdrücken, als auch für Programmiersprachen stellen sie sich als äusserst nützlich heraus.



University

Mit Bäumen kann man z.B. Hierarchien in Unternehmen abbilden. Nehmen wir eine Hochschule. Dort gibt es einen Präsidenten, für jede Fakultät einen Dekan, und dann gibt es da noch Professoren und Tutoren. Zunächst legen wir für jedes Mitglied der Hierarchie einen Knoten an.

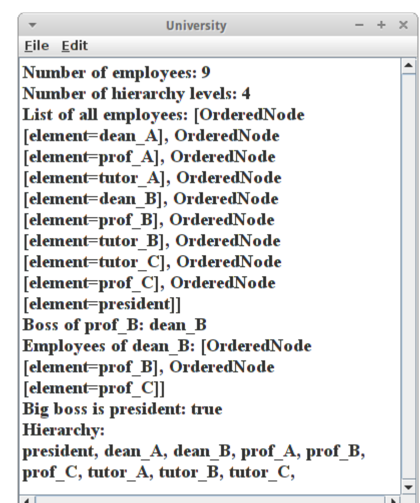
```
OrderedNode<String> president = new
OrderedNode<String>("president");
OrderedNode<String> dean_A = new OrderedNode<String>("dean_A");
OrderedNode<String> dean_B = new OrderedNode<String>("dean_B");
OrderedNode<String> prof_A = new OrderedNode<String>("prof_A");

...
```

Dann verbinden wir die Knoten miteinander:

```
dean_A.setParent(president);
...
prof_B.setParent(dean_A);
...
tutor_B.setParent(prof_B);

tutor_C.setParent(prof_B);
```

Und schließlich machen wir daraus einen Baum:

```
OrderedTree<String> university = new
OrderedTree<String>(president);
```



An den Baum können wir jetzt ein paar Fragen stellen. Z.B. wieviele Angestellte gibt es an der Uni:

```
println("Number of employees: " +
university.size());
```

Oder wieviele Hierarchieebenen gibt es:

```
println("Number of hierarchy levels: " +
university.height());
```

Wir können auch alle Angestellten auflisten die es an der Hochschule gibt:

```
println("List of all employees: " +
university.elements());
```

Wir können auch fragen, wer der Chef von *prof_B* ist:

```
println("Boss of prof_B: " + prof_B.getParent().getElement());
```

oder wer die Angestellten von *dean_B* sind:

```
println("Employees of dean_B: " + dean_B.getChildren());
```

wobei hier allerdings nur die direkten Kinder aufgelistet werden, die Enkelkinder, sprich Tutoren sind nicht dabei.

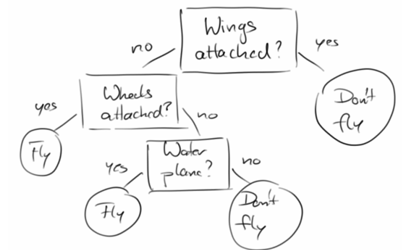
Wer der Oberboss ist könnten wir mit der *isRoot()* Methode herausfinden:

```
println("Big boss is president: " + president.isRoot());
```

Wenn wir noch alle Angestellten nach Hierarchieebenen sortiert ausgeben wollen, dann verwenden wir den Level-Order Traversal:

```
university.levelOrder(new VisitorInterface<String>() {
    public void visit(AbstractNode<?> node) {
        print(node.getElement() + ", ");
    }

});
```

PilotCheckList

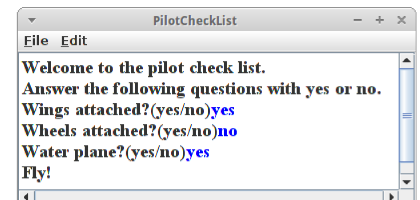


Bei Entscheidungsbäumen (decision trees) handelt es sich immer um Binärbäume. Als einfaches Beispiel beginnen wir mit einer Pilotencheckliste, die jeder Pilot durchgehen sollte bevor er abhebt. Zunächst legen wir wieder den Baum an, dieses mal mit Hilfe der *BinaryTree* und *BinaryNode* Klassen:

```
BinaryNode<String> root = new
BinaryNode<String>("Wings attached?");
BinaryNode<String> wheels = new BinaryNode<String>("Wheels attached?");
root.setLeft(wheels);
...
BinaryTree<String> decisions = new BinaryTree<String>(root);
```

Dann folgt der sogenannt "Pilot Walk Through": beginnend von der Wurzel des Baums, gehen wir eine Frage nach der anderen durch, und je nachdem ob der Nutzer mit Ja oder Nein antwortet, navigieren wir durch den Baum nach links oder rechts:

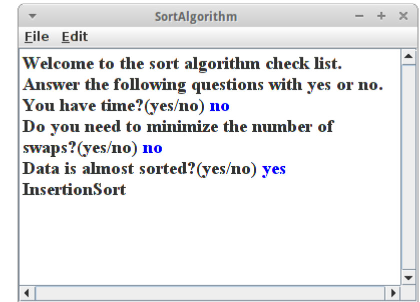
.




```
private void pilotWalkThrough() {
    println("Welcome to the pilot check list.");
    println("Answer the following questions with
yes or no.");
    // start with the root:
    BinaryNode<String> currentNode =
 (BinaryNode<String>) decisions.root();
    while (currentNode.isInternal()) {
        String answer = readLine(currentNode.getElement() + "(yes/no)");
        if (answer.equals("yes")) {
            currentNode = currentNode.getLeft();
        } else {
            currentNode = currentNode.getRight();
        }
    }
    println(currentNode.getElement());
}
```

Am Ende geben wir dann die Empfehlung zu fliegen oder nicht zu fliegen.

SortAlgorithm



Machen wir noch ein weiteres Beispiel für einen Entscheidungsbaum: Eine Empfehlung für die richtige Wahl eines Sortieralgorithmuses. Vor ein oder zwei Kapiteln haben wir folgendes gelernt:

- BubbleSort dauert sehr lange, also nehmen wir es nur, wenn wir viel Zeit haben.
- Wenn wir die Anzahl der Swaps minimieren müssen, dann verwenden wir SelectionSort.
- Wenn unsere Eingabedaten fast sortiert sind, dann ist InsertionSort die beste Wahl.
- Für alles andere verwenden wir QuickSort.

Daraus machen wir jetzt einen Entscheidungsbaum. Allerdings wollen wir dieses Mal einen Parser verwenden, genauer gesagt den *BinaryTreeParser*:

```
decisions = new BinaryTreeParser().parseTree(new
File("sort_algorithm.txt"));
```

Der macht aus einer Textdatei einen Baum. Dabei muss die Textdatei aber gewissen Regeln folgen. Hier ist die Datei für unsere Sortieralgorithmen:

```
You have time? {
 BubbleSort,
 Do you need to minimize the number of swaps? {
  SelectionSort,
  Data is almost sorted? {
   InsertionSort,
   QuickSort
  }
 }

}
```

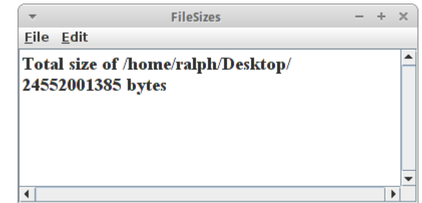
Das Wurzel-Element ist die erste Frage, also "You have time?". Kinder beginnen mit einer geschweiften Klammer '{' und werden danach aufgelistet, mehrere Kinder werden durch Komma ',' getrennt. Bei Binärbäumen kann es höchstens zwei Kinder geben, dabei ist das erste Kind für den Ja-Fall und das zweite Kind für den Nein Fall. Natürlich muss jede geöffnete Klammer wieder geschlossen werden. Der Walk-Through selbst ist komplett identische wie im vorherigen Beispiel.

Noch mehr Decision Trees

Man kann aus fast allem Entscheidungsbäume machen. Hier sind ein paar Vorschläge:

- "So You Need A Typeface" von Julian Hansen ist ein Entscheidungsbaum mit dessen Hilfe man die richtige Schriftart für seine Anwendung findet. [1]
- Sollte eine Person einen Kredit bekommen? [2]
- Welche Programmiersprache soll ich für mein Projekt verwenden? [3]
- Für die Datenvisualisierung können wir zwischen Balkendiagramm, Kreisdiagramm, Baumdiagramm, Linienplot, Streudiagramm, etc. wählen. In dem Buch *Data Points* von Nathan Yau [4] kann man ab Seite 137 Kriterien nachlesen, wie man die passende Visualisierung auswählt. Daraus kann man einen Entscheidungsbaum machen.
- Welche Game-Engine soll ich verwenden? Man könnte Kosten, Lizenz, Support, Anzahl der Projekte, etc. als Kriterien verwenden.
- Was soll ich studieren? Überlegen Sie sich Fragen, die es einem Benutzer helfen zu entscheiden, was er oder sie studieren sollte.
- Sogar Schach kann man als Entscheidungsbaum darstellen. [5]

Man könnte sich auch Fragen überlegen mit denen man z.B. Vögel, Bäume, Tiere, Pflanzen, Krankheiten, usw. klassifizieren könnte. Weiter unten werden wir uns damit beschäftigen wie man Wildbienen identifiziert.



FileSizes

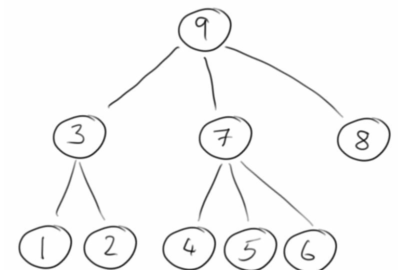
Im Kapitel über Rekursion haben wir schon einmal die Summe der Größe aller Dateien in einem Verzeichnis ermittelt. Wir wollen das jetzt noch einmal machen aber dieses mal mit Hilfe eines Baumes. Wir beginnen damit, dass wir die Verzeichnisse rekursiv durchlaufen:

```
private void buildFileTreeRecursive(OrderedNode<File> node) {
    File file = node.getElement();
    if (file.listFiles() != null) {
        for (File child : file.listFiles()) {
            // we are only interested in directories
            if (child.isDirectory()) {
                OrderedNode<File> newDir = new OrderedNode<File>(child);
                node.addChild(newDir);
                buildFileTreeRecursive(newDir);
            }
        }
    }

}
```

Unser *OrderedNode<File>* enthält dieses Mal *Files* anstelle von Strings. Das ist ganz praktisch wie wir gleich sehen werden. Diese Methode rufen wir von der *createFileTree()* Methode auf:

```
private OrderedTree<File> createFileTree(File startFile) {
    OrderedNode<File> directories = new OrderedNode<File>(startFile);
    buildFileTreeRecursive(directories);
    return new OrderedTree<File>(directories);

}
```

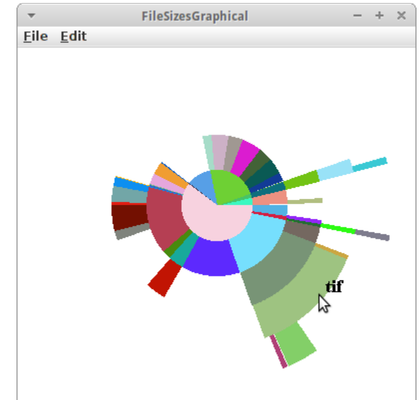
So jetzt haben wir den Baum, was machen wir damit? Oben beim Post-Order Traversal haben wir gehört, dass man damit die Gesamtgröße aller Dateien in einem Verzeichnis, aber auch jeweils aller seiner Unterverzeichnisse ermitteln kann. Das machen wir jetzt:




```
OrderedTree<File> fileTree = createFileTree(new
File("/home/ralph/"));

fileTree.postOrder(new VisitorInterface<File>() {
    @Override
    public void visit(AbstractNode<?> node) {
        long sizeOfFilesInDir = 0;
        File f = (File) node.getElement();
        if (f.isDirectory()) {
            for (File ff : f.listFiles()) {
                if (ff.isFile()) {
                    sizeOfFilesInDir += ff.length();
                }
            }
        } else {
            println("we should never get here!");
        }
```

```
            totalSize += sizeOfFilesInDir;
            //println(node +": "+sizeOfFilesInDir);
        }

    });
```

Mit der postOrder() Methode gehen wir also den ganzen Baum rekursiv durch. Wir ermitteln also zunächst die Größe von Knoten '1', dann die von '2', danach können wir die von '3' ausrechnen. Das machen wir solange bis wir bei der Wurzel angekommen sind.



FileSizesGraphical

Dass sich die ganze Arbeit lohnen kann, sehen wir jetzt: wir wollen alle Verzeichnisse und ihre Größen in einer Art Kuchendiagramm graphisch darstellen. Im Prinzip haben wir im Projekt *FileSizes* schon fast alles was wir brauchen. Was fehlt ist, dass wir uns irgendwie merken können was denn die Größe aller Dateien in einem gewissen Unterverzeichnis war. Im Projekt *FileSizes* haben wir die einfach weggeschmissen. Der Trick ist anstelle der Klasse File eine eigene Klasse zu definieren:

```
class FileAndSize {
    public File file;
    public long size;

    public FileAndSize(File file, long size) {
        this.file = file;
        this.size = size;
    }

}
```

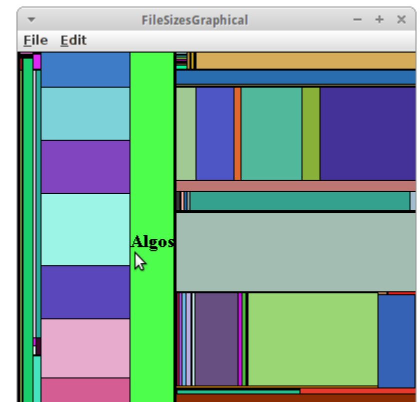
Das ist jetzt nicht ganz koscher, weil die Instanzvariablen public sind, da das aber eine lokale Klasse ist, geht das schon in Ordnung. Damit sind die beiden Methoden *buildFileTreeRecursive()* und *createFileTree()* vom obigen Beispiel fast identisch, wir müssen lediglich *File* durch *FileAndSize* ersetzen und den Konstruktoraufruf durch

```
OrderedNode<FileAndSize> directories =

    new OrderedNode<FileAndSize>(new FileAndSize(startFile, -1));
```

Wir erhalten dann also einen Tree der aus Verzeichnissen mit ihren jeweiligen Größen besteht:

```
OrderedTree<FileAndSize> fileTree = createFileTree(new
File("/home/ralph/"));

traverseAndDrawArcs(fileTree.root(), 0, 360, 0);
```

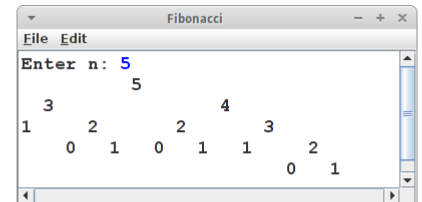
Diesen Baum durchlaufen wir jetzt rekursiv und zeichnen GArcs mit einer Breite die jeweils proportional zur Verzeichnisgröße ist. Das geht automatisch von außen nach innen, da es ja ein rekusiver Algorithmus ist.




```
private void
traverseAndDrawArcs(AbstractNode<FileAndSize> tree,
        double alpha, double sweep, int depth) {
    // base case
    if (depth > 8)
        return;

    // recursive case
    depth++;
```

```
        long parentSize = tree.getElement().size;
        for (AbstractNode<FileAndSize> child : tree.getChildren()) {
            long childSize = child.getElement().size;
            double deltaAlpha = sweep * childSize / parentSize;

            if (deltaAlpha > 1) {
                traverseAndDrawArcs(child, alpha, deltaAlpha, depth);
                int radius = depth * DISK_THICKNESS;
                GArc arc =
                    new GArc(centerX - radius / 2, centerY - radius / 2,
                             radius, radius, alpha, deltaAlpha);
                arc.setFilled(true);
                arc.setColor(rgen.nextColor());
                add(arc);
            }
            alpha += deltaAlpha;
        }

    }
```

Schaut ganz hübsch aus. Man kann das auch mit GRects machen, ist aber nicht ganz so hübsch.



Fibonacci

Wir haben ja vor ein paar Kapiteln die Bekanntschaft des Herren Fibonacci gemacht. Das man seine Zahlen auch als Baum darstellen kann wollen wir jetzt zeigen. Wobei das mit dem Baum sehr wörtlich gemeint ist. Der Fibonacci Baum ist ein binärer Baum, und er besteht aus Zahlen, deswegen verwenden wir *BinaryNode<Integer>* für unsere Knoten als Datentyp. Wir verwenden wieder unsere rekursive Methode *fibu()* übergeben aber anstelle von Zahlen einen BinaryNode:

```
    private int fibo(BinaryNode<Integer> node) {
        int n = node.getElement();
        switch (n) {
        case 0:
            return 0;
        case 1:
            return 1;
        default:
            BinaryNode<Integer> left = new BinaryNode<Integer>(n - 2);
            node.setLeft(left);
            BinaryNode<Integer> right = new BinaryNode<Integer>(n - 1);
            node.setRight(right);
            return fibo(right) + fibo(left);
        }

    }
```

Daraus machen wir dann einen BinaryTree:

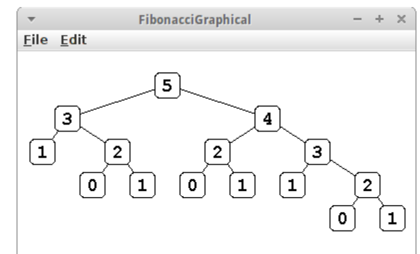
```
    BinaryNode<Integer> root = new BinaryNode<Integer>(n);
    fibo(root);

    BinaryTree tree = new BinaryTree<Integer>(root);
```

Wie stellen wir nun den Baum dar? Dafür verwenden wir die Hilfsklasse *TreePrinter*, die uns ein String Array zurückgibt,

```
String tmp[][] = new tree.TreePrinter().prettyPrintSimple(tree);

printTreeVertical(tmp);
```

welches wir dann einfach nur noch in unserem ConsoleProgram ausgeben müssen:

```
private void printTreeVertical(String[][] tmp) {
    for (int i = 0; i < tmp.length; i++) { // 4
        for (int j = 0; j < tmp[0].length; j++) { // 7
            System.out.println();
            if (tmp[i][j] != null) {
                print(tmp[i][j] + " ");
            } else {
                print("  ");
            }
        }
        println();
    }

}
```



FibonacciGraphical

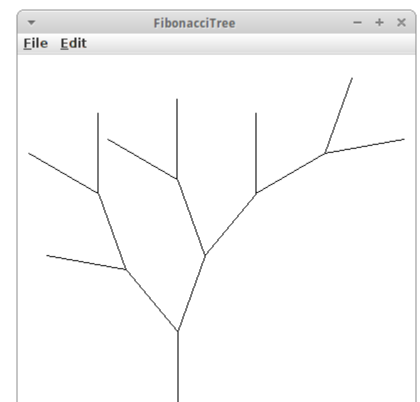
Die Textausgabe von Bäumen, so wie wir es oben gemacht haben ist nicht sehr befriedigend. Viel schöner wäre es wenn man Bäume graphisch ausgeben könnte. Mit dem *BinaryTreeDrawerCanvas* geht das sogar relativ einfach. Der Code ist fast identisch mit dem obigen. Natürlich muss unser Programm jetzt ein *Program* sein, also

```
public class FibonacciGraphical extends Program
{ ... }
```

und wir müssen den Baum der *BinaryTreeDrawerCanvas* Klasse übergeben, die wir dann wie aus dem ersten Semester gewohnt zu unserer UI hinzufügen:

```
BinaryTree<Integer> tree = new BinaryTree<Integer>(root);
canvas = new BinaryTreeDrawerCanvas(tree);
canvas.setShapeNode(BinaryTreeDrawerCanvas.SHAPE_ROUNDRECT);
canvas.setOrientation(BinaryTreeDrawerCanvas.VERTICAL);
canvas.setEdgeStyle(BinaryTreeDrawerCanvas.EDGE_STYLE_DIRECT);
canvas.setFont("Courier new-bold-18");

add(canvas, CENTER);
```

Auch wieder viel hübscher.

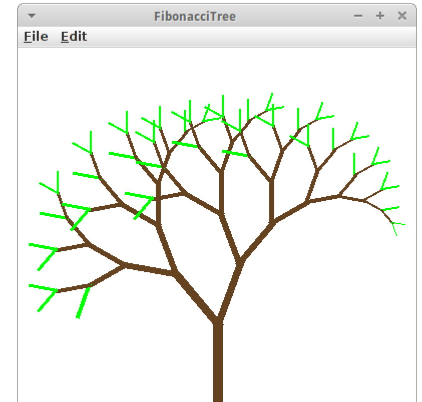


FibonacciTree

Dass man mit den Fibonacci Zahlen und dem Pre-Order Traversal wirklich Bäume zeichnen kann, wollen wir jetzt zeigen. Wir verwenden wieder unsere *fibu()* Methode von oben, um unseren Binärbaum zu erzeugen.

```
private double len = 80;
private double leftAngle = -40;
private double rightAngle = 20;

public void run() {
    int n = 5;
    BinaryNode<Integer> root = new
```

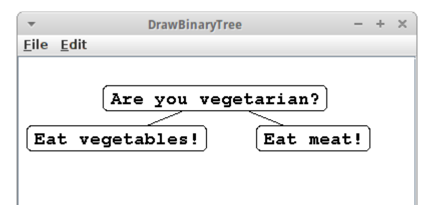
```
BinaryNode<Integer>(n);
    fibo(root);
    preOrderSimple(root, SIZE/2-40, SIZE-44, -Math.toRadians(90));

}
```



Wir haben auch noch drei Instanzvariablen eingeführt, eine für die Länge der "Zweige", eine für den Winkel um den sich die Äste nach links lehnen sollen, und einen Winkel für die rechten Äste. Der Pre-Order Traversal sieht dann wie folgt aus:

```
private void preOrderSimple(BinaryNode<Integer>
node, int x0, int y0, double alpha) {
    //visit
    GLine line = new GLine(x1, y1, x1 + len *
Math.cos(alpha), y1 + len * Math.sin(alpha));
    add(line);
    x0 += len * Math.cos(alpha);
    y0 += len * Math.sin(alpha);

    // recurse
    if (node.hasLeft()) {
        preOrderSimple(node.getLeft(), x0, y0,
alpha+Math.toRadians(leftAngle));
    }
    if (node.hasRight()) {
        preOrderSimple(node.getRight(), x0,
y0,alpha+Math.toRadians(rightAngle));
    }

}
```

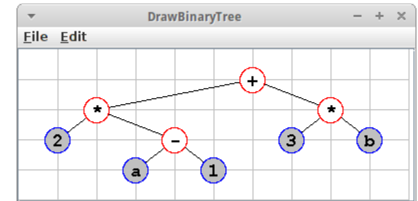
Mit ein paar kleinen Modifikationen wird das dann ganz ansehnlich und man kann sogar eine gewisse Ähnlichkeit mit Bäumen nicht verleugnen.



DrawBinaryTree

Um Bäume einigermaßen ansprechend darzustellen gibt es die TreeDrawerCanvas und zwar in zwei Versionen, einmal für binären Bäume, *BinaryTreeDrawerCanvas*, und dann auch für beliebige Bäume, *OrderedTreeDrawerCanvas*. Verwendet werden beide fast identisch,

```
public class DrawBinaryTree extends Program {
    public void init() {
        BinaryTree<String> tree =
            new BinaryTreeParser().parseTree(new File("eat.txt"));
        BinaryTreeDrawerCanvas canvas = new BinaryTreeDrawerCanvas(tree);
        canvas.setNodeSeparationX(98);
        add(new JScrollPane(canvas), CENTER);
    }

}
```



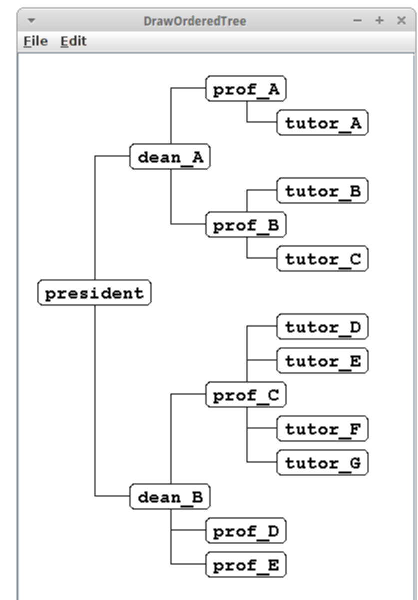
Das Beispiel zeigt einen binären Bäum an, der aus der Datei "eat.txt" erzeugt wurde. Wir können verschiedene Parameter variieren, z.B., können wir bei den Shapes wählen zwischen

- SHAPE_OVAL
- SHAPE_RECT
- SHAPE_ROUNDRECT

Ausserdem können wir die Farben der internen und externen Knoten setzen, auch deren Hintergrundfarbe:

```
canvas.setShapeNode(BinaryTreeDrawerCanvas.SHAPE_OVAL);
canvas.setColorExternalNodes(Color.BLUE);
canvas.setColorExternalNodesFill(Color.LIGHT_GRAY);
canvas.setColorInternalNodes(Color.RED);

canvas.setGridOn(true);
```

Wir können auch ein Hintergrundgitter anzeigen lassen.



DrawOrderedTree

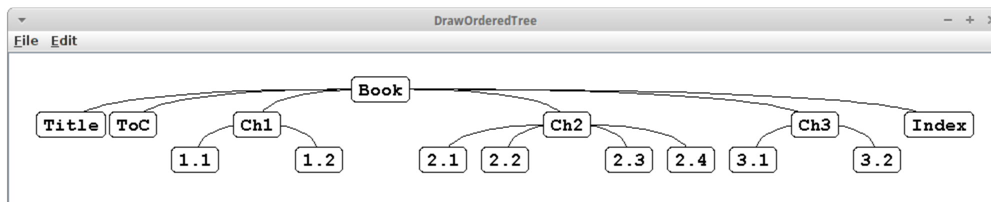
Geordnete Bäume können im Unterschied zu binären Bäume auch mehr als zwei Kinder haben. Ein Beispiel ist unser Hochschulbeispiel vom Anfang des Kapitels. Es macht Sinn diesen Baum von links nach rechts, also horizontal darzustellen. Und für die Verbindungslinien, Edges, haben wir einen anderen Stil gewählt:

```
canvas.setOrientation(
    OrderedTreeDrawerCanvas.HORIZONTAL);
canvas.setEdgeStyle(
    OrderedTreeDrawerCanvas.EDGE_STYLE_SQUARE);
```

Wenn wir die Verbindungen zwischen den Knoten lieber mit sogenannten "Quad-Line" Segmenten verbinden möchten,

```
canvas.setEdgeStyle(
    OrderedTreeDrawerCanvas.EDGE_STYLE_QUAD);
```

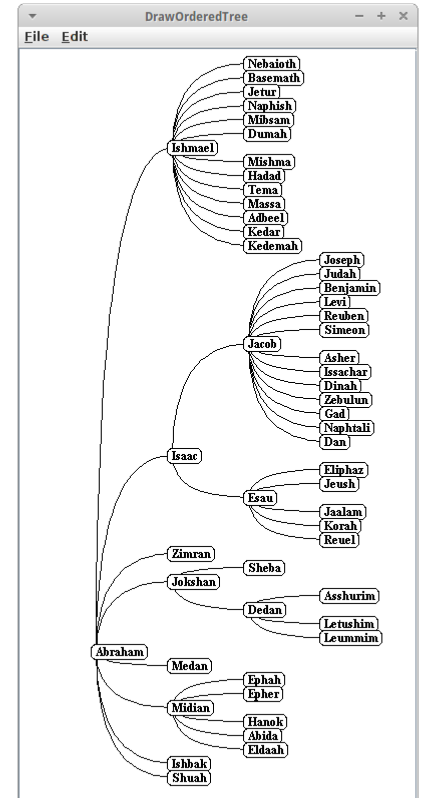
dann eignet sich das recht gut für die Darstellung von Büchern:



.

Für Genealogien, eignet sich vielleicht wieder die horizontale Auflistung, allerdings machte es hier Sinn, das Alignment anzupassen:

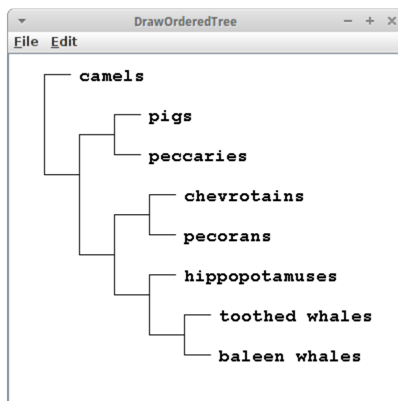
```
canvas.setOrientation(
    OrderedTreeDrawerCanvas.HORIZONTAL);
canvas.setAlignment(
    OrderedTreeDrawerCanvas.ALIGN_LEFT);
canvas.setEdgeStyle(
    OrderedTreeDrawerCanvas.EDGE_STYLE_QUAD);
```



Möchte man aber z.B. phylogenetische Beziehungen zwischen Säugetieren darstellen, dann will man nur die externen Knoten darstellen, interne Knoten sollen unsichtbar bleiben. Die Verbindungslinien sollen wieder rechtwinklig sein, und man möchte keine Umrandung sehen.

```
canvas.setEdgeStyle(
    OrderedTreeDrawerCanvas.EDGE_STYLE_SQUARE);
canvas.setShowInternalNodes(false);
canvas.setColorExternalNodes(Color.WHITE);
```

Das sieht dann so aus:



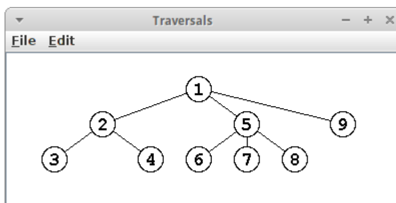
Ich denke das dürfte genügen, um zu sehen, dass wir wahrscheinlich 80% aller Bäume halbwegs o.k. aussehend darstellen können.

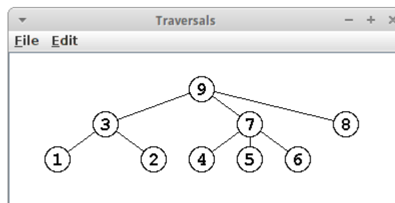
Traversals

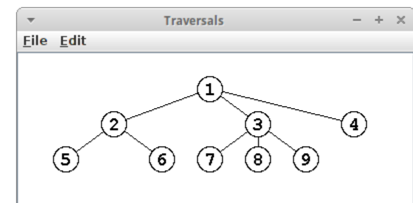
Ein anderes hübsches Beispiel ist es einmal alle different Traversals auszuprobieren. Der folgende Code erlaubt das an einem einfachen Beispiel:

```
String str = "A{B{C,D},E{F,G,H},I}";
OrderedTree<String> tree = new OrderedTreeParser().parseTree(str);

//tree.levelOrder(new VisitorInterface<String>() {
//tree.postOrder(new VisitorInterface<String>() {
tree.preOrder(new VisitorInterface<String>() {
    int counter = 1;
    public void visit(AbstractNode<?> node) {
        ((OrderedNode<String>)node).setElement(""+counter++);
    }

});
```

Die Zahl gibt die Reihenfolge an in der die jeweiligen Knoten besucht wurden, als erstes für Pre-Order, gefolgt von Post-Order und schließlich Level-Order Traversal:







Pre-Order Post-Order Level-Order

Parsing

In dem *SortAlgorithm* Projekt haben wir einen Parser verwendet, genauer gesagt den *BinaryTreeParser*:

```
decisions = new BinaryTreeParser().parseTree(new
File("sort_algorithm.txt"));
```

Wie funktioniert der? Eigentlich haben wir schon einige Parser selbst geschrieben, denn jedesmal wenn wir den *StringTokenizer* verwendet haben, haben wir irgendetwas geparst. Schauen wir uns erst einmal an was wir parsen wollen:

```
Are_you_vegetarian? {
 Eat_vegetables!,
 Eat_meat!
}
```

Daraus wollen wir einen Binärbaum parsen. Die Trennzeichen sind die geschweiften Klammer '{' und '}' und das Komma ','. Also verwenden wir

```
StringTokenizer st = new StringTokenizer(line, "{},", true);
```

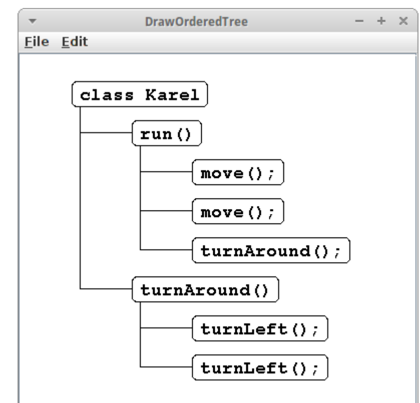
dabei ist das letzte *true* ganz wichtig, denn es besagt, dass auch die Trennzeichen in Tokens verwandelt werden. Wir schleifen dann einfach durch

```
while (st.hasMoreTokens()) {
    String token = st.nextToken().trim();
    if (token.length() > 0) {
    ...
    }
}
```

und den Rest der Arbeit erledigen wir mit einem *switch*:

```
switch (token) {
case "{":
    ...
    break;
case "}":
    ...
    break;
case ",":
    ...
    break;
default:
    ...

    break;
```

Und das ist das Geheimnis eines Parsers. Die können wir auch selber schreiben.



Java

Beginnen wir mit unserem ersten Parser, und zwar einen für Java. Wenn wir uns das folgende einfach Java Programm ansehen,

```
class Karel {
  run() {
    move();
    move();
    turnAround();
  }
  turnAround() {
    turnLeft();
    turnLeft();
  }

}
```

dann sehen wir, dass das eigentlich ein Baum ist. Der Tokenizer ist der Gleiche wie eben, nur das Komma ersetzen wir durch einen Strichpunkt:

```
StringTokenizer st = new StringTokenizer(line, "{};", true);
```

Wir gehen wie oben einen Token nach dem andern durch und müssen uns nur ein bisschen überlegen was bei den geschweiften Klammer passieren soll:

```
switch (token) {
case "{":
    classLevel = !classLevel;
    break;
case "}":
    currentNode = (OrderedNode<String>) currentNode.getParent();
    classLevel = !classLevel;
    break;
case ";":
    break;
default:
    if (classLevel ) {
        OrderedNode<String> tmp = new OrderedNode<String>(token);
        currentNode.addChild(tmp);
        currentNode = tmp;
```

```
        } else {
            currentNode.addChild(new OrderedNode<String>(token));

        }
        break;

    }
```

Wir haben eine lokale Variable *classLevel* eingeführt, die vor der *while* Schleife definiert wird:

```
    boolean classLevel = false;
```

Wenn wir jetzt die erste geschweiften Klammer sehen, dann wird *classLevel* auf true gesetzt. Im *default* Zweig heißt das, dass wir es jetzt mit einer neuen Methodendefinition zu tun haben. Wenn wir danach der zweiten geschweiften Klammer begegnen, dann wird *classLevel* auf false gesetzt, d.h. wir sind jetzt innerhalb einer Methode. Dass bedeutet, dass wir im *default* Zweig einfach ein Statement nach dem anderen zu unserem *currentNode* als Kinder hinzufügen. Kommt jetzt eine schließende geschweifte Klammer, dann verlassen wir den Methoden-Modus und sind wieder zurück im Klassen-Modus, also *classLevel* ist true. Zusätzlich gehen wir aber wieder hoch in unserem Baum, deswegen die Zeile

```
    currentNode = (OrderedNode<String>) currentNode.getParent();
```

Jetzt kann entweder die nächste Methode kommen, oder wir sind fertig. Das ist jetzt nicht der tollste Parser auf der Welt, aber er zeigt das Prinzip. Und wir sollten nicht vergessen, wir sind erst im zweiten Semester!

Was wir jetzt haben ist ein *Abstract Syntax Tree* (AST) [6,7]. Das ist super-cool, denn damit könnten wir jetzt entweder

- den Code ausführen oder
- wieder Code erzeugen.

Das zweite hört sich jetzt erst mal bescheuert an: wir haben doch gerade erst aus dem Java den AST gemacht. Wieso sollte jetzt irgend jemand aus dem AST wieder Java machen wollen? Wer hat denn gesagt, dass es Java sein muss? Es könnte fast jede beliebige, objekt-orientierte Sprache sein, z.B. C++, Python oder JavaScript.

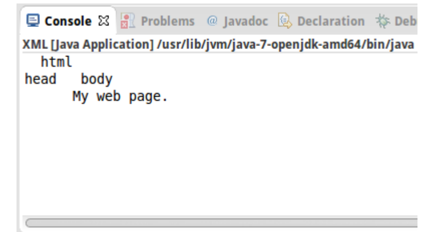
Da wir aber nur Java können, machen wir wieder Java daraus. Das geht ganz einfach mit einer abgewandelten Version des Level-Order Traversals:

```
    private String java = "";
    private void levelOrder(AbstractNode<String> node, int level) {
        if (node == null) {
            return;
        }
        if (level == 2) {            // class level
            java += node.getElement() + " {\n";
            for (AbstractNode<String> child : node.getChildren()) {
                levelOrder(child, level - 1);
            }
            java += "}\n";
        } else if (level == 1) {    // method level
            java += " " + node.getElement() + " {\n";
            for (AbstractNode<String> child : node.getChildren()) {
                levelOrder(child, level - 1);
            }
            java += " }\n";
        } else if (level == 0) {    // statement level
            java += "  " + node.getElement() + ";\n";
        }

    }
```

Wenn wir diese Methode mit unserem AST aufrufen,

```
String javaCode = levelOrder(tree.root(), 2);
```

dann kommt da Java raus. Cool, oder?



XML

Als nächstes wollen wir HTML parsen. HTML, eine spezielle Form von XML, ist nämlich auch ein Baum. Unser einfaches HTML sieht wie folgt aus:

```
<html>
  <head>
  </head>
  <body>
    My web page.
  </body>
</html>
```

Unser Parser müsste also folgende Trennzeichen erkennen: '<', '>' und '/'. Das erledigen wir mit

```
OrderedNode<String> root = null;
OrderedNode<String> currentNode = null;

StringTokenizer st = new StringTokenizer(line, "<>/", true);
```

Wir benötigen wieder die *while* Schleife und das *switch* wie oben, das ist einfach. Aber wir müssten irgendwie zwischen den öffnenden Tags, z.B. <body>, und den schließenden Tags, z.B. </body>, unterscheiden können. Das können wir über eine Zustandsvariable *state* machen,

```
String state = "";  // "open", "close", "normal"
```

die in den drei Zuständen "open", "close" oder "normal" sein kann. Zwischen den Zuständen hin- und herschalten tun wir über die Trennzeichen:

```
switch (token) {
case "<":
    state = "open";
    break;
case ">":
    state = "normal";
    break;
case "/":
    state = "close";
    break;
default:
    ...
}
```

Alles was jetzt noch zu tun bleibt, ist im *default* Zweig den Baum zusammenzubauen. Das geht wieder über einen *switch*, dieses mal abhängig vom Zustand:

```
switch (state) {
case "open":
    if (root == null) {
        root = new OrderedNode<String>(token);
        currentNode = root;
```

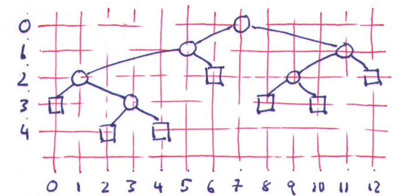
```
        } else {
            OrderedNode<String> node = new OrderedNode<String>(token);
            currentNode.addChild(node);
            currentNode = node;
        }
        break;
    case "close":
        currentNode = (OrderedNode<String>) currentNode.getParent();
        break;
    default:
        OrderedNode<String> node = new OrderedNode<String>(token);
        currentNode.addChild(node);
        break;

    }
```

Das war's. Damit wir checken können ob das auch richtig ist, können wir über die Hilfsklasse *TreePrinter()*,

```
OrderedTree<String> tree = new OrderedTree<String>(root);

new TreePrinter().prettyPrintSimpleVertical(tree);
```

eine Textversion des Baums auf der Konsole ausgegeben lassen. Eine Anmerkung: wenn wir schon *Regular Expressions* kennen würden, dann wäre das Parsen von HTML Pipifax. Was wir hier gebaut haben ist unsere erste *State-Machine*. War doch gar nicht so schwer.

Könnten wir aus dem HTML Baum auch wieder HTML generieren? Wie wäre es mit Level-Order Traversal?



ArithmeticExpression

Inzwischen haben wir die Auswertung arithmetischer Ausdrücke schon zweimal gesehen: einmal im Zusammenhang mit der Stack Klasse, und dann im Zusammenhang mit Rekursion. Hier wollen wir das Problem mit einem binären Baum lösen. Dabei geht es uns um drei Dinge:

- einen String wie z.B. "3+(4*5)" zu parsen,
- mittel In-Order Traversal aus dem Baum wieder einen arithmetischer Ausdruck zu machen und
- via Post-Order Traversal den arithmetischen Ausdruck auszuwerten, also zu berechnen.

Parsing

Das Parsen ist der etwas kompliziertere Teil. Wir beginnen mit der Definition eines Stacks der BinaryNodes enthält:

```
Stack<BinaryNode<String>> stackOfNodes = new Stack<BinaryNode<String>>();
```

Mit dem StringTokenizer zerlegen wir dann einen Ausdruck wie "3+(4*5)" in seine Einzelteile:

```
StringTokenizer st = new StringTokenizer(ariExp, " ()+-*/", true);
```

aus denen wir dann, je nach Token, eine Baumstruktur zusammenbauen:

```
while (st.hasMoreTokens()) {
    String tok = st.nextToken();
    switch (tok) {
    case "(":
    case " ":
        // do nothing
        break;
```

```
        case ")":
            BinaryNode<String> operand1 = stackOfNodes.pop();
            BinaryNode<String> operator = stackOfNodes.pop();
            BinaryNode<String> operand2 = stackOfNodes.pop();
            operator.setLeft(operand2);
            operator.setRight(operand1);
            stackOfNodes.push(operator);
            break;
        default:
            BinaryNode<String> node = new BinaryNode<String>(tok.trim());
            stackOfNodes.push(node);
            break;
        }

    }
```

Bei genügenden Klammern sind wir jetzt fertig. Sollten aber nicht genügend Klammern gesetzt worden sein, müssen wir unseren Stack noch etwas aufräumen:

```
    while (stackOfNodes.size() > 1) {
        BinaryNode<String> operand1 = stackOfNodes.pop();
        BinaryNode<String> operator = stackOfNodes.pop();
        BinaryNode<String> operand2 = stackOfNodes.pop();
        operator.setLeft(operand2);
        operator.setRight(operand1);
        stackOfNodes.push(operator);
    }

    return stackOfNodes.pop();
```

Das letzte Element auf dem Stack ist unser gesuchter Binärbaum. Es sei angemerkt, dass unser Parser etwas dumm ist, mit Punkt vor Strich hat er es nicht so.

Print

Aus dem binären Baum wieder einen arithmetischer Ausdruck zu machen geht ganz einfach via In-Order Traversal:

```
    printExpression(p) {
        if hasLeft(p) {
            print("(");
            printExpression( left(p) )
        }
        print( p.element() )
        if hasRight(p) {
            printExpression( right(p) )
            print(")");
        }

    }
```

Evaluate

Die Auswertung ist eine abgewandelte Version des Post-Order Traversals. In Pseudo-Code sieht das so aus:

```
    int evaluate(p) {
        if isExternal(p) {
            return( p.element() )
        } else {
            x = evaluate( leftChild(p) )
            y = evaluate( rightChild(p) )
```

```
            O <- operator stored at p
            return( x O y )
        }

    }
```

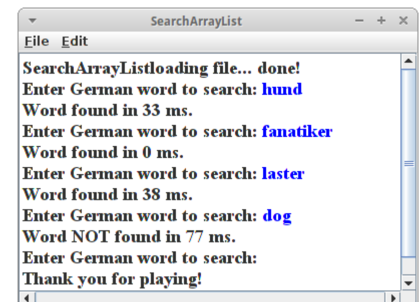
Erinnern wir uns, in den externen Knoten sind die Zahlen gespeichert, deswegen wenn wir einen externen Knoten haben, geben wir einfach den Wert des Knotens zurück.

Wenn wir einen internen Knoten haben, dann muss es sich um einen Operator handeln, also +, -, * oder /. Zu jedem Operator muss es aber zwei Kinder geben, deswegen besuchen wir erst einmal die Kinder rekursiv, und wenden dann danach (Post) die gewünschte Operation auf die beiden Kinder an. Das war's.

SearchBinaryTree

In diesem Projekt wollen wir uns ein bisschen mit der binären Suche mit Hilfe eines binären Baumes beschäftigen. Als Suchbeispiel nehmen wir unser Englisch-Deutsches Wörterbuch, *dictionary_en_de.txt*. Dabei interessieren uns aber nur die deutschen Wörter.

SearchArrayList



Zum Aufwärmen verwenden wir eine ganz normale ArrayList. Wir schreiben also eine Klasse *SearchArrayList*, mit folgender Instanzvariable:

```
private List<String> al = new ArrayList<String>();
```

Die befüllen wir in einer *loadLexiconFromFile()* Methode mit deutschen Wörtern aus dem Wörterbuch:

```
...
StringTokenizer st = new StringTokenizer(words, "=");
String en = st.nextToken();
String de = st.nextToken();

al.add(de.toLowerCase().trim());
```

Und dann benutzen wir die *contains()* Methode um festzustellen ob ein Wort in der Liste ist oder nicht:

```
...
boolean found = false;
long startTime = System.currentTimeMillis();
for (int i = 0; i < 10; i++) {
    found = al.contains(searchWord.toLowerCase().trim());
}

long endTime = System.currentTimeMillis();
```

Das Ganze machen wir zehn mal, um aussagekräftige Zeiten zu bekommen. Wir stellen hier fest:

- Im Durchschnitt dauert es so 30 ms um ein Wort zu finden.
- Wörter die am Anfang sind, wie "Fanatiker", werden sehr schnell gefunden.
- Wörter die am Ende sind, wie "Laster", dauern etwas länger.
- Wörter die nicht in der Liste sind dauern auch länger.

Wenn wir uns überlegen wie die Suche in einer Liste funktioniert, ist das genau was wir erwarten.

SearchBinaryTree

Versuchen wir es jetzt zum Vergleich mit einem binären Suchbaum. Für den Baum brauchen wir einen Knoten, wir nennen ihn *SearchBinaryNode*, der folgende Attribute haben soll:

```
public class SearchBinaryNode<E> {
    private SearchBinaryNode<E> leftChild;
    private SearchBinaryNode<E> rightChild;
    private E element;

    // constructor and methods...

}
```

Damit der Knoten nützlich ist, müssen wir folgende Methoden implementieren:

- SearchBinaryNode(E element)
- E getElement()
- boolean hasLeft()
- boolean hasRight()
- SearchBinaryNode<E> getLeft()
- SearchBinaryNode<E> getRight()
- setLeft(SearchBinaryNode<E> child)
- setRight(SearchBinaryNode<E> child)

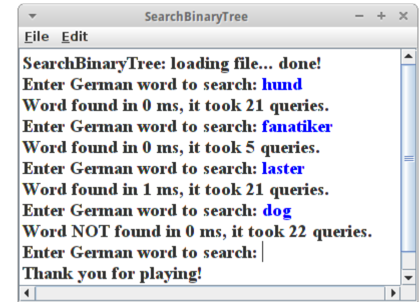
Die Namen der Methoden sind ziemlich selbsterklärend. Wir können die jetzt schnell implementieren, oder wir verwenden die *tree.BinaryNode* Klasse.

Die eigentliche Suche implementieren wir dann in der Klasse *SearchBinaryTree*. Wir benötigen wieder eine *loadLexiconFromFile()* Methode. In der rufen wir eine Methode *add()* auf die wir schreiben müssen:

```
public void add(String word) {
    if (root == null) {
        root = new SearchBinaryNode<String>(word);
    } else {
        SearchBinaryNode<String> current = root;

        // first find if word is already in tree

        // insert element if not already in tree

    }

}
```

Für die Suche müssen wir noch eine *contains()* Methode schreiben:

```
public boolean contains(String word) {
    SearchBinaryNode<String> current = root;
    while (true) { // loop and a half
        String x = current.getElement();
        int comparison = x.compareTo(word);
        if (comparison == 0) {
            // we found it
            return true;
        } else if (comparison > 0) {
            // go to right
            ...
        } else {
            // go to left
            ...
        }
    }
    return false;

}
```

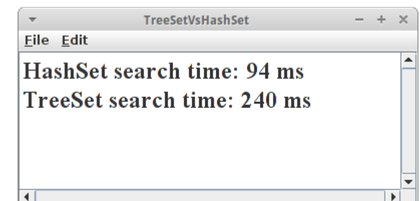


Wieder machen wir die Suche zehn mal, um aussagekräftige Zeiten zu bekommen. Wir stellen hier fest:

- Im Durchschnitt dauert es so 0 ms um ein Wort zu finden.
- Es macht keinen Unterschied ob Wörter am Anfang oder am Ende sind, oder überhaupt in der Liste.

Interessant ist auch wieviele Vergleiche notwendig sind: bei unserem Tree sind das im Schnitt so zwischen 5 bis 22 Vergleiche um ein Wort zu finden. Bei der Liste sind das viel mehr: wenn ein Wort am Ende der Liste ist benötigen wir ca. 100000 Vergleiche! Da ist er wieder unser Freund der Logarithmus!

Wir sehen also, dass Bäume unsere Freunde sind. Allerdings ein Wort der Vorsicht: was passiert wenn wir unser Programm mit den englischen Wörter ausprobieren? Da die Wörter in der Datei *dictionary_en_de.txt* alphabetisch sortiert sind, führt das dazu, dass unser Baum nicht mehr symmetrisch ist. Was wiederum dazu führt, das unsere Suche langsamer wird. Im schlimmsten Fall genauso langsam wie mit der Liste. Um dieses Problem, der "unbalanced trees" zu vermeiden gibt es ganz viel Forschung, und deswegen gibt es auch so viele verschieden Baumarten, Red-Black Trees wären z.B. eine gute Wahl.



TreeSetVsHashSet

Der Vorteil des TreeSet ist, dass es sortiert ist, der des HashSets, dass es schneller ist. Um genau zu sein, sowohl Einfügen als auch Suchen ist logarithmisch beim TreeSet, O(log n), während es beim HashSet konstant ist, O(1). Wir können das mit einem kleinen Programm messen (ähnlich wie bei ArrayListVsLinkedList). Wichtig ist allerdings, dass wir nicht nur eine Zahl in die Sets einfügen, sondern viele verschiedene, zufällige.

```
for (int i = 0; i < 10000000; i++) {
    int randomPos = (int) (100000.0 * Math.random());
    set.add(randomPos);

}
```

Danach messen wir dann die Zeit die benötigt wird um zu testen ob eine beliebige Zahl in dem Set enthalten ist:

```
long start = System.currentTimeMillis();
for (int i = 0; i < 1000000; i++) {
    // read an element at a random position:
    int randomPos = (int) (100000.0 * Math.random());
    st.contains(randomPos);
}
long end = System.currentTimeMillis();

System.out.println(end - start);
```

Das tun wir einmal für eine HashSet und einmal für eine TreeSet.

Challenges

Wildbienen in Deutschland

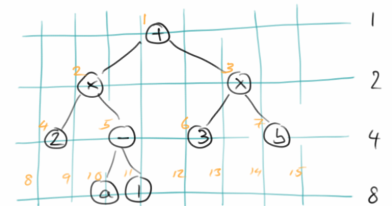
Den 560 Wildbienenarten in Deutschland geht es nicht so gut, deswegen sollen wir ein Programm schreiben, dass es ermöglicht sieben der häufiger vorkommenden Wildbienen zu identifizieren. Dazu müssen wir erst einmal den Flyer "WILDBIENEN" der Initiative "Deutschland summt!" herunterladen [8]. Auf Seite 5 dieses Flyers sehen Sie eine Tabelle mit dem Namen "Wildbienen bestimmen - leicht gemacht!". Daraus machen wir einen Entscheidungsbaum.

Entscheidungsbaum (Decision Tree)

Machen Sie aus dieser Tabelle einen Entscheidungsbaum (decision tree), am einfachsten malen Sie sich diesen auf einem Stück Papier auf. Achten Sie darauf, dass es ein Binärbaum ist, mit Ja/Nein (yes/no) Entscheidungen.

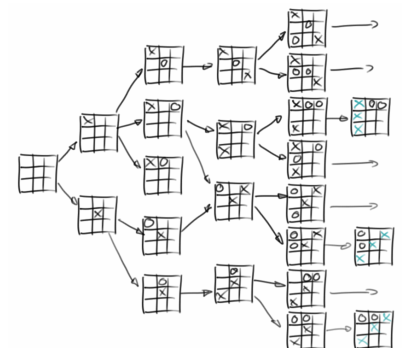
Erstellung des Programms

Orientieren Sie sich am Beispiel "PilotCheckList", das demonstriert wie ein Entscheidungsbaum befüllt wird, wie man diesen auf der Konsole ausgibt (printTreeNicely), und wie man damit eine Benutzerbefragung durchführt (pilotWalkThrough). Versuchen Sie diesen Code zu verstehen und dann in der Klasse WildBienen ein Bestimmungsprogram für WildBienen zu schreiben.



Array based BinaryTree

Unsere Implementierung für den BinaryTree verwendet eine Linked Binary Tree Struktur. Man kann BinaryTrees aber auch mit einem Array bzw. einer ArrayList abbilden. Dazu muss man sich lediglich die Skizze rechts ansehen. Dann erkennt man, dass es in jeder Generation n maximal 2^n Knoten geben kann. Ausserdem erkennt man, dass man die Positionen der möglichen Kinder vorhersagen kann. Das ist alles was man braucht um die Datenstruktur umzusetzen.



Game Trees and Tic-Tac-Toe

Zunächst lesen wir den Artikel in der Wikipedia bzgl Game Trees [9]. Danach sollten wir uns den noch interessanteren Artikel von Victor S.Adamchik zum gleichen Thema ansehen [10]. Mit der Information, kann man sich jetzt überlegen wie man einen Game Tree für das Spiel Tic-Tac-Toe schreibt.

Research

Auch in diesem Kapitel kann man wieder ein bisschen was erforschen.

Visualization

Mit Jason Park's Algorithm Visualizer [11] lassen sich die verschiedensten Algorithmen und auch Datenstrukturen sehr schön visualisieren. Wir sollten uns mal die Beispiele zu Bäumen ansehen.

QuadTree

In vielen Spielen hat man es sehr häufig mit Kollisionserkennung zu tun. Z.B. in unserem Asteroids Spiel aus dem ersten Semester geht es darum Kollisionen mit Asteroiden zu erkennen. Aber auch im Agrar Beispiel ist Kollisionserkennung ganz wichtig. Vor allem bei sehr vielen Objekten kann das dazu führen dass unser Program super-langsam wird, wenn wir nicht die richtige Datenstruktur verwenden. In zwei Dimensionen ist das der QuadTree [12], in drei der Octree [13]. Auch k-d Trees [14] werden in diesem Zusammenhang oft eingesetzt. Wir wollen uns ein bisschen inn die Materie einlesen.

Other Trees

Wie bereits angedeutet gibt es noch ganz viele andere Bäume. Z.B. sind da:

- AVL Trees
- Splay Trees
- (2,4) Trees
- Red-Black Trees
- B-Trees

Wir sollten uns mal zu all den Bäumen schlau machen, vor allem aber den Red-Black Trees und den AVL Trees.

Eine interessante Frage ist z.B., wann sollten wir einen Red-Black Tree, wann einen AVL Tree und wann eine B-Tree verwenden?

- Red-Black Trees und AVL Trees sind binäre Bäume, B-Trees hingegen können mehr als zwei Kinder pro Zweig haben, sind also nicht so tief.
- Red-Black Trees nehmen es mit dem Re-Balancing nicht so ernst, was Inserts und Deletes schneller sein lässt.
- AVL Trees sind etwas genauer mit dem Re-Balancing, was dazu führt, dass sie schneller beim Lesen sind.

Fragen

1. Zeichnen Sie einen einfachen Baum. An diesem Beispiel zeigen Sie die folgende Begriffe:
    - Root
    - Internal Node
    - External Node
    - Ancestors
    - Descendants
    - Subtree

2. Ist der Post-Order Traversal Algorithmus ein rekursiver Algorithmus?

3. Betrachten Sie den folgenden Baum und betrachten Sie den Knoten 'B':
    - Was sind seine Ancestors?
    - Was sind seine Descendants?
    - Was ist die Depth des Knotens B?
    - Was ist die Height des Baums?

4. Was ist der Unterschied zwischen einem binären Baum und einem gewöhnlichen Baum?

5. Erklären Sie den Unterschied zwischen Pre-Order Traversal und Post-Order Traversal eines Baumes.

6. Betrachten Sie den folgenden arithmetischen Ausdruck und zeichnen Sie den dazu gehörigen binären Baum.
    - (2 * (a - 1) + (3 * b))
    - (4 - (x - 1)) + (2 * y)
    - (6 – 3) * (4 % 5) + 6)
    - 3 * ((4 % 5) * (6 – 3))

7. Zeichnen Sie den binären Suchbaum, der aus dem Einfügen der folgenden zehn Zahlen resultiert:
    - {16, 24, 8, 30, 42, 25, 1, 9, 18, 29}.
    - {24, 8, 16, 30, 1, 9, 42, 25, 18, 29}.
    - {29, 18, 25, 42, 9, 1, 30, 16, 8, 24}.

Referenzen

Referenzen kommen und gehen. Hoffen wir, dass die hier etwas länger halten. Leider scheint der Website zum Buch "Data Structures and Algorithms" von Bruno R. Preiss im Moment nicht mehr erreichbar zu sein. Das Buch gibt's aber noch, hab's sofort gebraucht gekauft.

[1] *So You Need A Typeface* von Julian Hansen, www.julianhansen.com/#/zimmer/

[2] *Introduction of Decision Trees*, Bill Wilson, http://www.cse.unsw.edu.au/~billw/cs9414/notes/ml/06prop/id3/id3.html

[3] *Choosing a programming language*, Brett Slatkin, www.onebigfluke.com/2016/01/choosing-a-programming-language.html

[4] *Data Points: Visualization That Means Something*, Nathan Yau, John Wiley & Sons, 2013

[5] *Tree for chess*, https://c1.staticflickr.com/5/4017/4391267493_ab44e3f827_b.jpg

[6] *Abstract syntax tree*, https://en.wikipedia.org/wiki/Abstract_syntax_tree

[7] *CSE 2231: Software II: Software Development and Design*, http://web.cse.ohio-state.edu/software/2231/web-sw2/extras/slide/21.Abstract-Syntax-Trees.pdf

[8] *WILDBIENEN*, Initiative "Deutschland summt!", http://www.deutschland-summt.de/?file=files/media_ds/pdfs/2017/Wildbienen_Folder_23-03-2017.pdf

[9] Game tree, https://en.wikipedia.org/w/index.php?title=Game_tree&oldid=740804477 (last visited Feb. 23, 2017).

[10] Game Trees, Victor S.Adamchik, www.cs.cmu.edu/~adamchik/15-121/lectures/Game%20Trees/Game%20Trees.html

[11] Algorithm Visualizer, Jason Park, http://algo-visualizer.jasonpark.me/

[12] Why algorithms matter. Quad tree example, javaprogrammernotes.blogspot.de/2015/01/why-algorithms-matter-quad-tree-example.html

[13] Octree, https://en.wikipedia.org/wiki/Octree

[14] k-d tree, https://en.wikipedia.org/wiki/K-d_tree

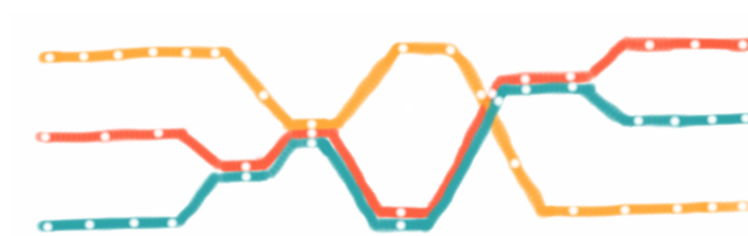
[15] Decision tree, https://en.wikipedia.org/wiki/Decision_tree

[16] Encoding general trees as binary trees, en.wikipedia.org/wiki/Binary_tree

[17] Abraham's family tree, https://en.wikipedia.org/wiki/Abraham%27s_family_tree

[18] Data Structures and Algorithms with Object-Oriented Design Patterns in Java by Bruno R. Preiss

Graphs

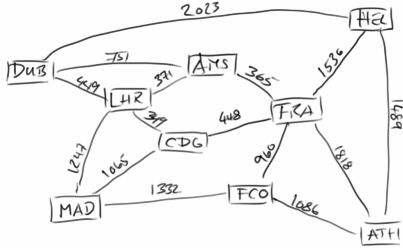


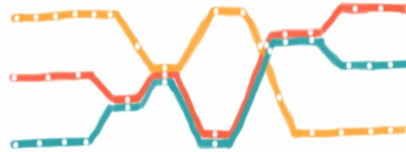
Graphen sind überall, sie sind sogar noch weiter verbreitet als Bäume. Sie sind auch viel nützlicher. Vieles was zunächst wie ein Baum aussieht ist in Wirklichkeit ein Graph. Z.B. der klassische Familien-Stammbaum, ist in der Regel ein Graph. Genauso verhält es sich bei den meisten Spielebäumen (z.B. Tic-Tac-Toe), oder auch dem Pascaldreieck, das zwar wie ein Baum aussieht aber ein Graph ist. Auch Mazes sind häufig keine Bäume, sondern Graphen.

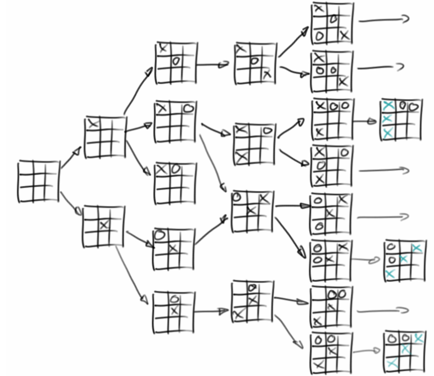
Bei den Algorithmen die mit Graphen zu tun haben, geht es meistens darum den kürzesten Pfad zu finden, oder herauszufinden ob ein Graph verbunden ist oder Kreise enthält. Aber auch für die Planung eines Straßennetzwerkes gibt es Graphalgorithmen. Ganz wichtig sind Graphen auch bei der Planung von Projekten.

Examples

Der Klassiker unter den Graphenanwendungen ist Google Maps. Aber egal ob Auto, Zug oder Flugzeug immer wenn man wissen möchte ob es einen Weg von A nach B gibt, oder was der kürzeste Weg von A nach B ist, dann verwendet man Graphenalgorithmen. Die Züge eines Spiel, wie z.B. Tic-Tac-Toe, stellen bei genauem Betrachten eine Graphstruktur und nicht eine Baumstruktur dar.





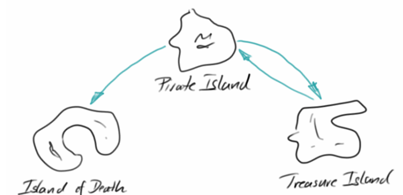


Weitere Beispiele für Graphen sind:

- Freundesnetzwerke oder Beziehungen zwischen Menschen im Allgemeinen
- das Internet und all seine Webseiten
- Computer Netzwerke
- elektrische Schaltkreise
- Transport Netzwerke
- Planung von Tasks
- Klassendiagramme
- Workflows

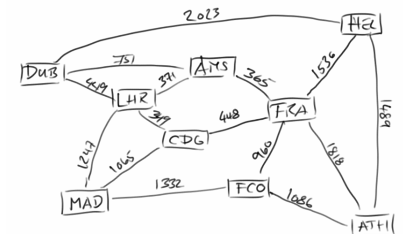
Es gibt noch unzählige andere Beispiele. Übrigens, alle Bäume sind auch Graphen, aber ein Graph ist nicht notwendigerweise auch ein Baum.

Pirate Island



Zur Einführung in Graphen betrachten wir eine Inselgruppe in der Karibik, bestehend aus den drei Inseln: *Pirate Island*, *Treasure Island* und *Island of Death* [1]. Die Inseln bezeichnen wir als *Vertices*. Zwischen den Inseln kann man mit Booten hin- und manchmal auch wieder zurückfahren. Diese Verbindungsstrecken nennen wir *Edges*. Edges können eine Richtung haben, also man kann nur in eine Richtung fahren, oder sie haben keine Richtung. Das Ganze, also die Liste aller Vertices und Edges nennt man einen *Graphen*.

Definitions



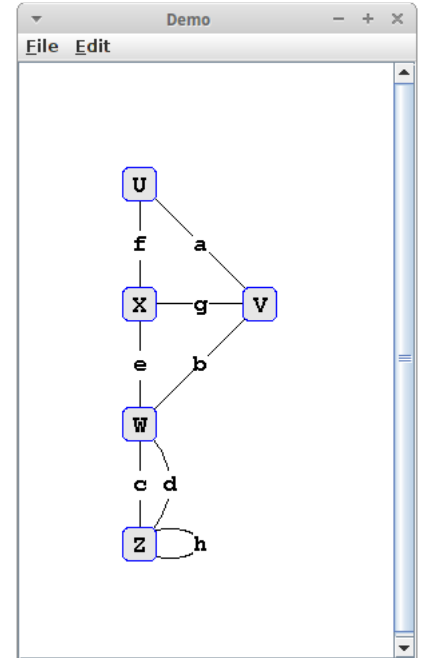
Auch was Graphen angeht, macht es Sinn sich erst einmal einen Wortschatz zuzulegen. Ein Graph besteht also aus Vertices und Edges.

- Ein Vertex ist ein Knoten in dem man Information speichern kann.
- Ein Edge verbindet immer zwei Vertices. Auch im Edge kann man Information speichern. Ein Edge kann gerichtet sein (directed), oder die Richtung ist egal (undirected).

Wenn wir z.B. Flughäfen mit ihren Verbindungsflügen betrachten, dann sind die Flughäfen die Vertices, und die Information die wir in den Vertices speichern ist der Name des jeweilgen Flughafens. Die Edges hingegen sind die Verbindungen die zwischen den Flughäfen existieren. Die Information die wir in den Edges speichern würden wäre dann die Entfernung zwischen den jeweilgen Flughäfen die durch dieses Edge verbunden werden.

Terminology

Wenn man allgemein von Graphen spricht, dann benutzt man Großbuchstaben aus dem Ende des Alphabets, wie U, V, W, usw. für Vertices und Kleinbuchstaben vom Anfang des Alphabets, wie a, b, c, usw. für Edges. Wenn wir den Graphen rechts betrachten, dann sagt man:



- *U* und *V* sind die "End-Vertices" des Edge *a*;
- *a*, *b*, und *g* sind "incident" auf *V*;
- *U* und *V* sind "adjacent", also benachbart;
- der Vertex *W* hat "degree" 4, weil er vier Edges hat;
- die Edges *c* und *d* sind "parallel";
- das Edge *h* ist ein "self-loop".

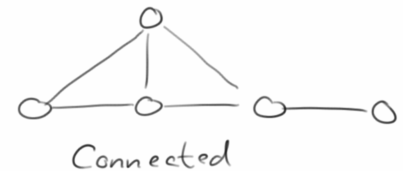
Durch einen Graph gibt es *Pfade*. Ein Pfad beginnt bei einem Anfangs-Vertex und endet bei einem End-Vertex:

```
P1 = (U,f,X,g,V,b,W)
```

dabei muss zwischen zwei Vertices immer ein Edge liegen, bestehen also abwechselnd aus Vertices und Edges.

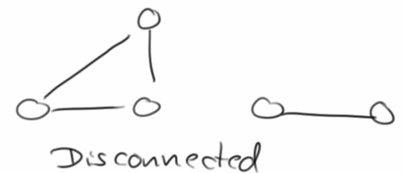
Besondere Pfade sind *Ringe* (cycles): Ringe sind Pfade bei denen Anfangs- und End-Vertex gleich sind:

```
C1 = (U,f,X,g,V,a,U)
```



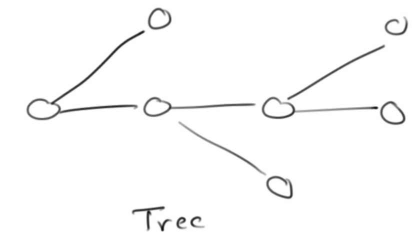
Ringe können problematisch sein, speziell für Navigationssysteme: sie führen u.U. dazu dass man immer im Kreis fährt ohne je an sein Ziel zu kommen. Auch für Datenpakete im Internet sind Ringe ein Problem.

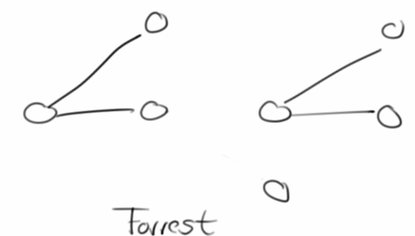
Wenn wir von einem *Subgraphen* eines Graphen *G* sprechen, dann enthält dieser eine Untermenge der Vertices und Edges von *G*. Ein Spezialfall ist der *Spanning Subgraph*: dieser enthält alle Vertices von *G*, nicht notwendigerweise aber auch alle Edges.



Wir nennen einen Graphen *connected* (verbunden), wenn es einen Pfad zwischen allen Vertices gibt.

Kommen wir zu den Bäumen (*Trees*): ein Baum ist eine Graph der connected (verbunden) ist, und der keine Ringe (Cycle) enthält. Bei einem *Forrest* (Wald) handelt es sich um einen Graphen der aus mehreren Bäumen besteht. Was uns schließlich zu einem ganz wichtigen Graphen führt, dem *Spanning Tree*: dabei handelt es sich um einen *Spanning Subgraph* der ein Baum ist.





Graph ADT

Graphen funktionieren etwas anders als Bäume. Während die Klasse *Tree* nicht viel mehr als ein Wrapper für den Wurzelknoten war, wird bei den Graphen die Hauptarbeit in der Klasse *Graph* selbst erledigt. Wir beschränken uns erst einmal auf ungerichtete Graphen. Um mit Graphen bequem arbeiten zu können, benötigen wir die folgenden Methoden zum Auflisten aller Edges und Vertices:

- **edges():** gibt uns alle Edges als Liste;
- **vertices():** gibt uns alle Vertices als Liste.

Zum Navigieren:

- **incidentEdges(Vertex<V> vertex):** gibt eine Liste aller Edges zurück, die mit dem Vertex *vertex* verbunden sind;
- **opposite(Vertex<V> vertex, AbstractEdge<E> edge):** gibt den Vertex zurück der über das Edge *edge* mit dem Vertex *vertex* verbunden ist;
- **endVertices(AbstractEdge<E> edge):** gibt die beiden Vertices zurück die über das Edge *edge* miteinander verbunden sind;
- **areAdjacent(Vertex<V> vertex1, Vertex<V> vertex2):** gibt *true* zurück, falls *vertex1* und *vertex2* direkt miteinander verbunden sind.

Zum Suchen:

- **containsVertex(Vertex<V> vertex):** stellt fest ob der gegebene Vertex im Graphen ist;
- **findVertex(V element):** sucht den Vertex, der das gegebene Element enthält;
- **containsEdge(AbstractEdge<E> edge):** stellt fest ob das gegebene Edge im Graphen ist;
- **findEdge(E element):** sucht das Edge, das das gegebene Element enthält.

Ganz nützlich sind die folgenden Hilfsmethoden:

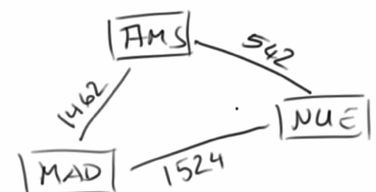
- **size():** gibt die Anzahl der Vertices im Graphen zurück;
- **isTree():** stellt fest ob es sich bei dem Graphen in Wirklichkeit um einen Baum handelt;
- **isConnected():** stellt fest ob der Graph verbunden (connected) ist;
- **hasCycle():** stellt fest ob der Graph Ringe (Cycles) enthält.

Bzgl. dem Einfügen, Ersetzen und Entfernen gibt es die folgenden Methoden:

- **insertVertex(Vertex<V> vertex):** fügt einen neuen Vertex in den Graphen ein;
- **insertEdge(Vertex<V> v1, Vertex<V> v2, E element):** fügt ein neues Edge in den Graphen ein;
- **replaceElement(Vertex<V> vertex, V element):** ersetzt das Element des Vertex *vertex*;
- **replaceElement(AbstractEdge<E> edge, E element):** ersetzt das Element des Edges *edge*;
- **removeVertex(Vertex<V> vertex):** entfernt den Vertex aus dem Graphen, entfernt auch alle Edges die mit diesem Vertex verbunden sind;
- **removeEdge(AbstractEdge<E> edge):** entfernt das Edge aus dem Graphen, die Vertices bleiben aber im Graphen.

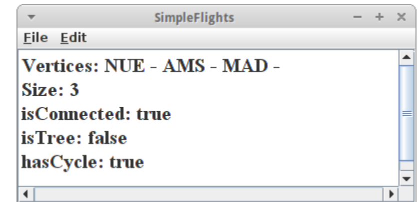
Ausserdem gibt es noch eine handvoll Traversal-Methoden, dazu weiter unten mehr.

Das ist jetzt ein bisschen viel auf einmal, deswegen sehen wir uns mal ein einfaches Beispiel an, wie man einen Graphen anlegt und verwendet:




```
GraphEdgeList<Integer, String> graph = new
GraphEdgeList<Integer, String>();
Vertex<String> v1 = graph.insertVertex(new
Vertex<String>("NUE"));
Vertex<String> v2 = graph.insertVertex(new
Vertex<String>("AMS"));
Vertex<String> v3 = graph.insertVertex(new Vertex<String>("MAD"));
AbstractEdge<Integer> e1 = graph.insertEdge(v1, v2, 542);
AbstractEdge<Integer> e2 = graph.insertEdge(v2, v3, 1462);
AbstractEdge<Integer> e3 = graph.insertEdge(v1, v3, 1524);
```

```
System.out.println("Size: " + graph.size());
System.out.println("isConnected: " + graph.isConnected());
System.out.println("isTree: " + graph.isTree());
System.out.println("hasCycle: " + graph.hasCycle());
```

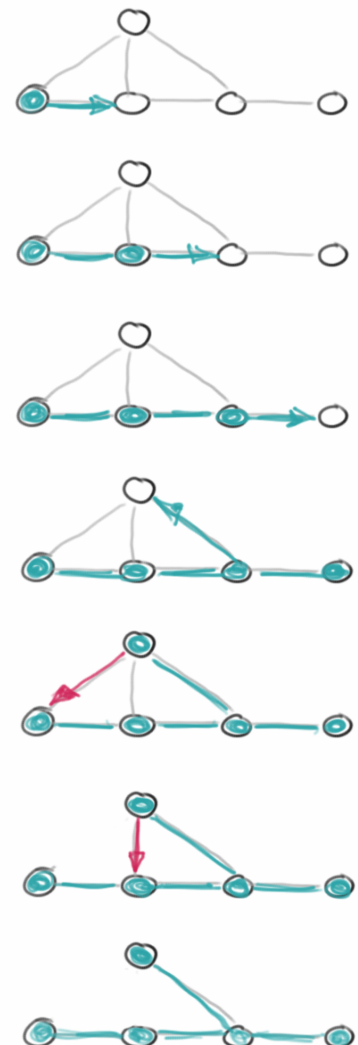


Wenn wir jetzt z.B. wissen wollten was die Entfernung zwischen dem Nürnberger Flughafen und dem Madrid Barajas International Airport ist, dann suchen wir erst einmal nach dem Nürnberger und Madrider Flughafen:

```
Vertex<String> v1 = graph.findVertex("NUE");
Vertex<String> v2 = graph.findVertex("MAD");

Collection<AbstractEdge<Integer>> connections = graph.incidentEdges(v1);
for (AbstractEdge<Integer> edge : connections) {
    Vertex<String> v3 = graph.opposite(v1, edge);
    if (v2 == v3) {
        println("Distance: "+edge.getElement());
    }
}
```

Danach müssen wir nach der Verbindungsstrecke (Edge) zwischen den beiden suchen. Das geht leider nicht direkt, aber wir können uns mal alle Verbindungen die von und nach Nürnberg gehen geben lassen, also *connections*. Dann iterierten wir einfach durch alle Verbindungen bis wir die finden an deren anderen Ende Madrid ist. Das war's.

Depth-First Search Traversal



Was die Iterationsmethoden angeht, sind die Tiefensuche (depth-first search, *DFS*) und die Breitensuche (breadth-first search, *BFS*) die wichtigsten. Beide durchsuchen den gesamten Graphen, besuchen alle Vertices und Edges, und können feststellen ob der Graph zusammenhängend (connected) ist. Man kann mit ihnen auch feststellen ob ein Graph Kreise enthält.

Die Tiefensuche markiert zunächste alle Vertices und Edges als "UNEXPLORED". Dann beginnt es mit einem beliebigen Vertex, markiert diesen als "VISITED" und folgt dann einem der Edges zum nächste Vertex. Dabei wird der neue Vertex mit "VISITED" markiert und das Edge mit "DISCOVERY". Dies wird dann rekursiv wiederholt. Sollte es passieren, dass wir unterwegs auf einen Vertex stoßen den wir schon einmal besucht haben, dann wird das Edge mit "BACK" markiert. Sobald es ein Edge gibt, das mit "BACK" markiert wurde, wissen wir, dass unser Graph Kreise enthält. Sollten wir im ersten Durchlauf alle Vertices besucht haben, dann wissen wir auch, dass der Graph zusammenhängend ist. Sollte es aber noch Vertices geben, die wir noch nicht besucht haben, müssen wir die natürlich auch noch besuchen.

Man kann diesen Algorithmus auch in sogenannten *Pseudo-Code* beschreiben, das macht es in der Regel einfacher die Übersicht zu behalten.

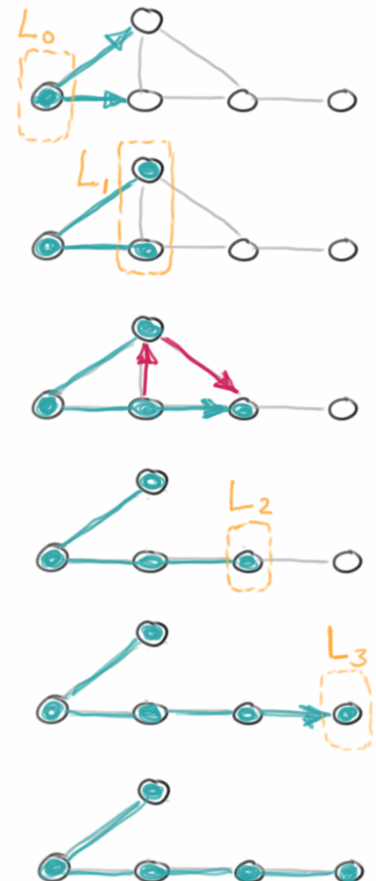
```
dfs(G) {
    for ( v in G.vertices() )
        setLabel( v, UNEXPLORED );
    for ( e in G.edges() )
        setLabel( e, UNEXPLORED );
    for ( v in G.vertices() )
        if ( getLabel( v == UNEXPLORED ) )
            dfs( G, v );

}
```

Wobei der rekurisve Teil wie folgt aussieht:

```
dfs(G, v) {
    setLabel( v, VISITED )
    for ( e in G.incidentEdges( v ) ) {
        if ( getLabel( e ) == UNEXPLORED ) {
            w = opposite( v, e );
            if ( getLabel( w ) == UNEXPLORED ) {
                setLabel( e, DISCOVERY );
                DFS( G, w );
            } else {
                setLabel( e, BACK );
            }
        }
    }
}
```

Der Algorithmus stammt aus dem wunderbaren Buch von Goodrich und Tamassia [2]. Das Depth-First Search Traversal entspricht dem Pre-Order Traversal bei Bäumen, d.h., er versucht zunächst so tief wie möglich in den Graph hineinzugehen, deswegen auch Tiefensuche.

Breadth-First Search Traversal



Die Breitensuche geht nicht in die Tiefe sondern die Breite. Auch hier markieren wir zunächste alle Vertices und Edges als "UNEXPLORED". Dann beginnen wir wieder mit einem beliebigen Vertex, markieren diesen als "VISITED" und fügen ihn in eine Liste *L0* ein. Dann folgen wir allen Edges die von diesem Vertex ausgehen, markiern sie als "VISITED" und fügen diese in eine Liste *L1* ein, also alle Vertices die ein Edge vom Ausgangsvertex entfernt sind. Zusätzlich markieren wir alle Edges denen wir gefolgt sind mit "DISCOVERY". Im dritten Schritt, folgen wir allen Edges die von den Vertices aus *L1* ausgehen. Alle Vertices die wir auf diese Art erreichen sammeln wir in der Liste *L2*. *L2* enthält also alle Vertices die zwei Edges vom Ausgangsvertex entfernt sind. Wir markieren auch wieder alle Edges mit "DISCOVERY", es sei denn sie führen zurück auf einen Vertex den wir bereits besucht haben, den markieren wir dann mit "CROSS". Das machen wir solange weiter, bis es nichts mehr zu entdecken gibt. Sollte es danach noch Vertices geben, die wir noch nicht besucht haben, müssen wir die natürlich auch noch besuchen.

Sollte es ein Edge mit "CROSS" geben, dann wissen wir wieder, dass unser Graph Kreise enthält. Auch hier gilt wieder, sollten wir im ersten Durchlauf nicht alle Vertices erreicht haben, dann ist der Graph nicht zusammenhängend.

Im *Pseudo-Code* sieht das dann so aus: erst markieren wir wieder alle Knoten und Edges:

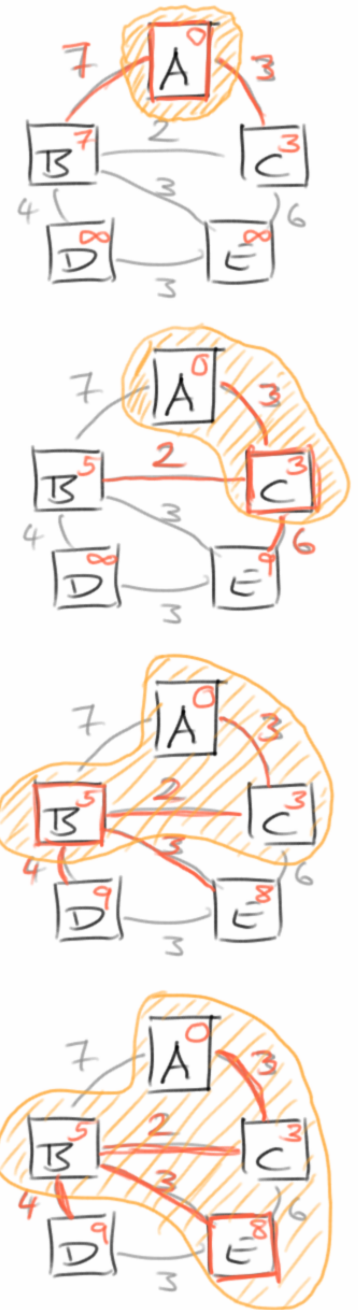
```
bfs(G) {
    for ( v in G.vertices() )
        setLabel( v, UNEXPLORED );
    for ( e in G.edges() )
        setLabel( e, UNEXPLORED );
    for ( v in G.vertices() )
        if ( getLabel( v == UNEXPLORED ) )
            bfs( G, v );
}
```

Und dann gehen wir alle durch,

```
bfs(G, s) {
    L_0 = new sequence;
    L_0.insertLast( s );
    setLabel( s, VISITED );
    i = 0;
    while ( !L_i.isEmpty() )
        L_i+1 = new sequence;
        for ( v in L_i.elements() ) {
            for ( e in G.incidentEdges( v ) ) {
                if ( getLabel( e ) == UNEXPLORED ) {
                    w = opposite( v, e );
                    if ( getLabel( w ) == UNEXPLORED ) {
                        setLabel( e, DISCOVERY );
                        setLabel( w, VISITED );
                        L_i+1.insertLast( w );
                    } else {
                        setLabel( e, CROSS );
                    }
                }
            }
        }
        i++;
    }

}
```

Auch dieser Algorithmus stammt aus dem wunderbaren Buch von Goodrich und Tamassia [2]. Der BFS Traversal Algorithmus entspricht dem Level-Order Traversal bei Bäumen. U.a. kann man mit ihm auch die Verbindung mit den wenigsten Edges zwischen zwei Knoten finden. Das muss aber lange nicht die kürzeste Verbindung sein.

Dijkstra



Kommen wir zur wichtigsten Anwendung von Graphen: wie komme ich am schnellsten von A nach B, also das typische Google Maps Problem. Aber auch die Datenpakete die im Internet unterwegs sind, sollten idealerweise auf dem kürzesten (oder schnellsten) Weg vom Server zum Browser geschickt werden.

Der bekannteste Algorithmus der das Problem des kürzesten Pfades (shortest path) effektiv löst stammt von Edsger Dijkstra [3]. Der Algorithmus geht davon aus, dass unser Graph zusammenhängend (connected) und ungerichtet (undirected) ist, und außerdem keine negativen Distanzen (weights) enthält. Es ist ein Greedy Algorithmus, der die Breitensuche (BFS) benutzt.

Wir beginnen beim Ausgangsvertex *A*, von dem wir unsere Reise beginnen möchten. Wir markieren alle Vertices mit der Distanz *Unendlich*, außer *A*, der wird mit der Distanz *0* markiert. Dann beginnen wir mit der Breitensuche und markieren alle Vertices die wir von *A* aus direkt erreichen können mit der Distanz die sie zu *A* haben. Jetzt kommt die Greedyness: wir machen mit dem Vertex *V* weiter, der die kürzeste Distanz zu *A* hat. Jetzt suchen wir nach allen Vertices die wir von diesem neuen Vertex *V* aus direkt erreichen können. Auch hier markieren wir wieder die Distanzen, allerdings die Distanz zum Ausgangsvertex *A*, nicht die zum neuen Vertex *V*. Hier gibt es zwei Alternativen: der Vertex ist noch mit *Unendlich* markiert, dann markieren wir ihn einfach mit der neuen Distanz zu *A*. Oder der Vertex war bereits markiert, da wir ihn schon in einem vorherigen Schritt erreicht haben. Dann markieren wir ihn nur dann neu, wenn die neue Distanz kürzer ist als die alte. Das wiederholen wir dann solange, bis wir alle Vertices im Graphen erreicht haben. Jeder Vertex ist jetzt mit der kürzesten Distanz zum Ausgangsvertex *A* markiert. Wenn wir uns noch zusätzlich merken, wer der Vorgängervertex war (prev) mit dem wir die letzte Änderung der Distanz vorgenommen haben, dann können wir auch den Pfad ausgeben, wie wir auf dem kürzesten Pfad von *A* nach *B* kommen.

In der Wikipedia findet man den Algorithmus auch als Pseudocode [4]:

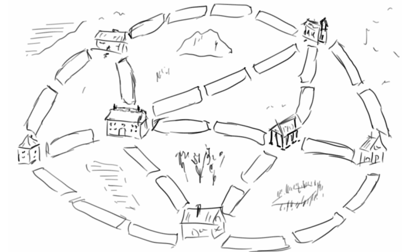
```
dijkstra(Graph, source):
    dist[source] ← 0
    create vertex set Q

    for each vertex v in Graph:
        if v ≠ source
            dist[v] ← INFINITY
            prev[v] ← UNDEFINED
        Q.add_with_priority(v, dist[v])

    while Q is not empty:
        u ← Q.extract_min()
        for each neighbor v of u:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)


    return dist[], prev[]
```

Vom Laufzeitverhalten her ist der Dijkstra AIgorithmus gar nicht so schlecht: O((n+m)*log(n)). Der A* Algorithmus [5] ist eine Erweiterung des Dijkstra AIgorithmus, die ein bisschen komplizierter ist, aber dafür auch schneller. Viele Spiele verwenden deswegen den A* Algorithmus anstelle von Dijkstra's.

Es gibt auch andere Algorithmen um den kürzesten Pfad zu bestimmen, bekannte sind:

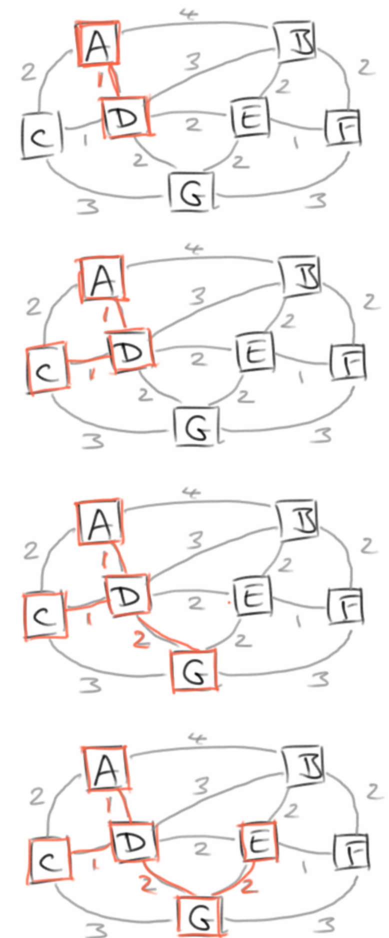
- Bellman-Ford: erlaubt auch negative Distanzen (weights) und läuft in O(n*m).
- DAG-basierte Algorithmen: funktioneren nur für gerichtete Graphen ohne Ringe, DAGs erlauben auch negative Distanzen (weights) und haben ein Laufzeitverhalten von O(n+m), sind also viel schneller als Dijkstra.

Minimum Spanning Tree



Kommen wir zur zweiten wichtigen Anwendung für Graphen: dem *Minimum Spanning Tree*. Erinnern wir uns an unsere Definitionen: ein *Spanning Tree* ist ein *Spanning Subgraph* der ein Baum ist. Der *Minimum Spanning Tree* ist der *Spanning Tree* mit der niedrigsten Gesamt-Gewichtung (total weight). Minimum Spanning Trees sind deswegen so wichig, weil sie uns helfen Geld zu sparen. Z.B. möchten wir bei der Planung von Straßen zwei Kriterien erfüllt sehen: zum Einen soll es zu jedem Dorf und jeder Stadt mindestens eine Straße geben. Zum Anderen möchte man aber auch Kosten sparen, also keine unnötigen Straßen bauen. Das Gleiche gilt auch für das Schienennetz. Oder wenn wir elektrische Leitungen verlegen, dann soll natürlich jedes Gerät mit Strom versorgt werden, aber Kabel kosten Geld, deswegen wollen wir es eigentlich vermeiden unnötige Kabel zu verlegen.

Prim



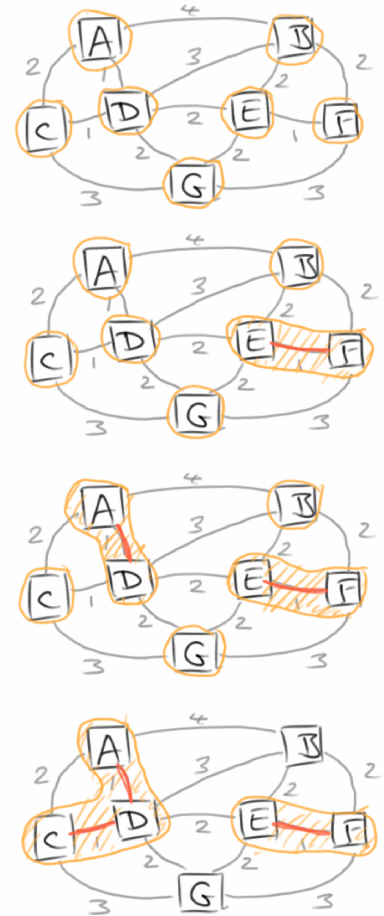
Das Problem lässt sich mit zwei bekannten Algorithmen lösen, dem von Prim und dem von Kruskal. In Prim's Algorithmus initialisieren wir zunächst eine Priorityqueue *Q* mit allen Vertices unseres Graphen, dabei markieren wir aber alle Vertices wieder mit *Unendlich*, außer unserem zufälligen Startvertex *r*. Die Priorityqueue verwendet die Distanz als Sortierkriterium. Dann beginnen wir damit aus der Priorityqueue uns das Element *u* mit der höchsten Priorität zu geben. Beim ersten mal ist das unser Startvertex *r*. Dann lassen wir uns alle Vertices geben, die man von *u* aus erreichen kann. Für all diese Vertices berechnen wir die Markierungen (Distanzen) zu *u*, aber nur wenn diese noch in *Q* sind. Danach nehmen wir das nächste Element aus der Priorityqueue und wiederholen den Prozess solange bis die Queue leer ist.

In der Wikipedia findet man den Algorithmus als Pseudocode [6,7]:

```
prim(G, r)
    Q: priority queue
    for ( v in G.vertices() )
        Q <- v

    for ( u in Q) )
        dist[u] <- ∞
        prev[u] <- null

    wert[r] <- 0
    while ( !Q.isEmpty() )
        u <- extract_min(Q)
        for ( all v ∈ neighbor(u) )
            if ( v ∈ Q  &&  weight(u,v) < dist[v] )
                prev[v] <- u
                dist[v] <- weight(u,v)
```

dabei ist weight() die Gewichtsfunktion für die Kantenlänge.

Kruskal



Kruskal's Algorithmus ist ein wunderbares Beispiel dafür, wie man einen eigentlich nicht ganz trivialen Algorithmus unter zuhilfenahme einer geschickt gewählten Datenstruktur ganz einfach erscheinen lassen kann. Besagte Datenstruktur ist in diesem Fall das Disjoint-Set (auf gut deutsch *Union-Find-Datenstruktur*) [9]. Das Disjoint-Set ist ein Set so wie wir es aus Kapitel 3 kennen, aber mit der Eigenschaft, dass es seine Unter-Sets mit verwaltet und zwar derart, das jedes Element in nur einem dieser Unter-Sets sein kann, aber auch jedes Element in einem Unter-Set sein muss. Diese Datenstruktur erlaubt es nun einmal mittels *find()* das Unter-Set zu finden dem ein bestimmtes Element angehört, und mittels *union()* zwei Unter-Sets zu vereinen.

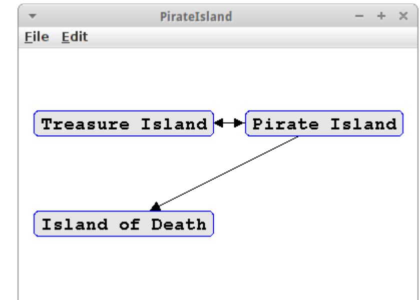
Kruskal's Algorithmus beginnt dann damit, dass er ein Disjoint-Set initialisiert, und jedes Element für sich erst einmal ein Unter-Set bildet. Wir initialisieren auch ein Minimum Spanning Tree Set, *A*. Das soll später alle Edges des Minimum Spanning Trees enthalten. Dann gehen wir durch alle Edges des Graphen, sortiert nach Distanz aufsteigend. Das macht man am einfachsten wieder mit einer Priorityqueue. Für die beiden Vertices *u* und *v* die zu dem jeweiligen Edge gehören lassen wir uns die jeweiligen Disjoint-Sets mittels *find()* geben. Sind diese beiden Disjoint-Sets verschieden, dann vereinigen wir diese mittels *union()* und fügen das Edge zu unserem Minimum Spanning Tree Set, *A*, hinzu. Das machen wir solange bis wir alle Edges durch sind.

Die Wikipedia enthält auch eine sehr schöne Erklärung des Kruskal Algorithmus und wie üblich findet man dort auch Pseudocode [8]:

```
kruskal(G)
    A = ∅   // set of edges
    foreach v ∈ G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E ordered by weight(u, v), increasing:
        if FIND-SET(u) ≠ FIND-SET(v):
            A = A ∪ {(u, v)}
            UNION(u, v)

    return A
```

Sieht doch gar nicht so kompliziert aus.

Directed Graphs (Digraph)



Gerichtete Graphen, auch Digraphs genannt, sind normale Graphen bei denen die Edges eine Richtung haben. Das wird graphisch meist durch einen Pfeil dargestellt. Digraphs haben zahlreiche Anwendungen. Z.B. die Links im Internet stellen einen gerichteten Graphen dar. Auch Flüge sind gerichtete Graphen. Es ist nicht selten, dass es einen Flug von A nach B gibt, aber es gibt keinen direkten Rückflug. Natürlich stellen auch Einbahnstraßen einen gerichteten Graphen dar. Im Prinzip kann jeder ungerichtete Graph in einen Digraph umgewandelt werden, indem man jedes ungerichtete Edge durch zwei gerichtete ersetzt, eines in die Hin- und eines in die Zurückrichtung.

Gerichtete Graphen haben alle Methoden eines normalen Graphen, zusätzlich gibt es aber noch folgende:

- **incomingEdges(Vertex<V> vertex):** gibt eine Liste aller Edges zurück, die mit dem Vertex *vertex* verbunden sind und deren Richtung auf den Vertex hin deutet;
- **outgoingEdges(Vertex<V> vertex):** gibt eine Liste aller Edges zurück, die mit dem Vertex *vertex* verbunden sind und deren Richtung von dem Vertex weg deutet.

Verwendet wird der Digraph ganz genauso wie der Graph. Wir implementieren das *Pirate Island* Beispiel:

```
DiGraphEdgeList<String, String> graph = new DiGraphEdgeList<String,
String>();

Vertex<String> v1 = graph.insertVertex(new Vertex<String>("Pirate
Island"));
Vertex<String> v2 = graph.insertVertex(new Vertex<String>("Island of
Death"));
Vertex<String> v3 = graph.insertVertex(new Vertex<String>("Treasure
Island"));

AbstractEdge<String> e1 = graph.insertEdge(v1, v2, "");
AbstractEdge<String> e2 = graph.insertEdge(v1, v3, "");

AbstractEdge<String> e3 = graph.insertEdge(v3, v1, "");
```

Auch nicht allzu kompliziert.

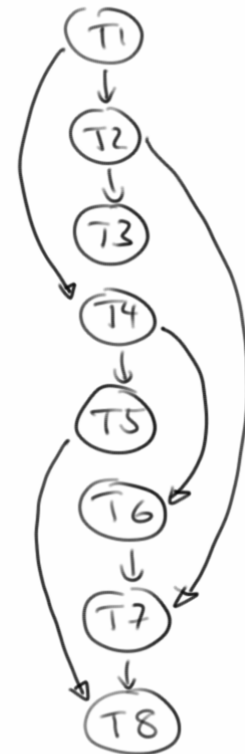
Die Traversal-Methoden verhalten sich ein bisschen anders für Digraphs. Es gibt sowohl ein gerichtetes DFS als auch ein gerichtetes BFS Traversal. Den Unterschied zu den normalen Traversal Methoden kann man sehr schön am *Pirate Island* Beispiel sehen: beginnt man seine Reise von *Treasure Island*, so kann man alle Inseln erreichen. Beginnt man allerdings im *Island of Death*, dann ist das nicht der Fall. Deswegen gibt es für gerichtete Graphen das Konzept der *Reachability*, also der Erreichbarkeit. Ein Digraph ist *strongly connected*, wenn von jedem Vertex aus jeder andere Vertex erreicht werden kann.

Directed Acyclic Graphs (DAG)

Eine ganz besondere Form der Digraphen sind die azyklischen: also gerichtete Graphen, die keine gerichteten Kreise enthalten. DAGs tauchen in sehr vielen interessanten Problemen auf [10], wie z.B. wenn wir in Java in einem Klassendiagramm Vererbungen darstellen, dann handelt es sich um einen DAG. Oder wenn wir die Abhängigkeiten von Vorlesungen betrachten, also man sollte erst Programmieren 1 gehört haben, bevor man Programmieren 2 hören kann, dann ist das auch wieder ein DAG. Die wichtigste Anwendung von DAGs sind allerdings Scheduling Constraints, wie sie z.B. im Projektmanagement auftreten.

Bei solchen Scheduling Constraints gibt es Abhängigkeiten. Z.B. können wir erst dann ein Bild aufhängen wenn wir vorher die Nägel gekauft haben, mit denen wir das Bild aufhängen wollen. Also ein Task hängt von einem anderen Task ab. Bei vielen Tasks, kann das etwas unübersichtlich werden. Um dann festzustellen, mit welchem Task man anfangen sollte, und in welcher Reihenfolge man die Tasks am effektivsten abarbeiten sollte, bedient man sich der *Topologischen Sortierung*.

Topological Sort



Topologische Sortierung hat damit zu tun, dass es einen Unterschied macht ob wir unsere Wäsche erst in die Waschmaschine und dann in den Trockner stecken oder umgekehrt. Im ersten Fall nämlich ist unsere Wäsche sauber und trocken, im zweiten Fall ist sie zwar auch sauber, aber nass. D.h. in unserem Graphen wollen wir sicher stellen, dass die Waschmaschine vor dem Trockner drankommt.

Es gibt mehrere Verfahren eine Topologische Sortierung durchzuführen. Die Resultate sind nicht immer die gleichen, da Topologische Sortierung nicht ganz eindeutig ist. Ein bekannter Algorithmus ist der von Kahn, den man in der Wikipedia als Pseudocode findet [11]:

```
L ← Empty list that will contain the sorted
elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

In den Projekten werden wir uns einige Beispiele ansehen.

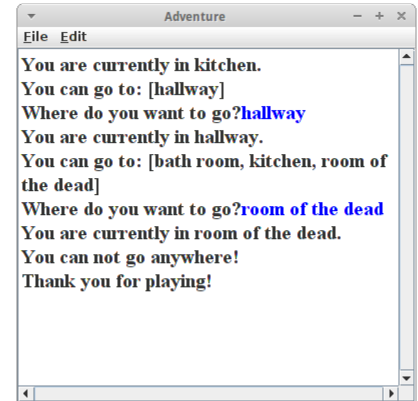
Review

Nach einigen Definitionen zu Graphen und deren Vertices und Edges, haben wir von gerichteten und ungerichteten Graphen gehört. Bzgl. Iterationsmöglichkeiten haben wir von der Tiefensuche (DFS) und der Breitensuche (BFS) gehört. Wir haben uns auch mit den Algorithmen zu Suche nach dem kürzesten Pfad (Dijkstra) und dem Minimum Spanning Tree (Prim und Kruskal) beschäftigt, und zum Schluss haben wir etwas von der topologischen Sortierung gehört.

Projekte

Die Projekte in diesem Kapitel haben es in sich: ob Abenteuerspiel, UBahn- und Flugnavigation, Stadt- und Projektplanung, nichts ist mehr vor uns sicher.

Adventure



Im dritten Kapitel haben wir das Adventure Spiel programmiert. Damals haben wir eine Map als Datenstruktur verwendet, genauer eine Map bestehend aus einem String als Key, dem Ausgangsraum, und einer Liste für die Räume, die man von diesem Ausgangsraum erreichen kann:

```
HashMap<String, ArrayList<String>> roomMap;
```

Im Prinzip ist nichts gegen diese Datenstruktur zu sagen, man kann das Problem aber auch mit einem Graphen lösen:

```
DiGraphEdgeList<String, String> diGraph;
```

Dabei sind die Räume die Vertices,

```
Vertex<String> v1 = diGraph.insertVertex(new Vertex<String>("kitchen"));

Vertex<String> v2 = diGraph.insertVertex(new Vertex<String>("hallway"));
```

und Räume werden über Edges miteinander verbunden:

```
AbstractEdge<String> e1 = diGraph.insertEdge(v1, v2, "");
```

Möchte man sein Abenteuer beginnen, so sucht man mit

```
Vertex<String> currentVertex = diGraph.findVertex("kitchen");
```

nach dem Anfangsvertex. Von diesem lässt man sich dann mittels

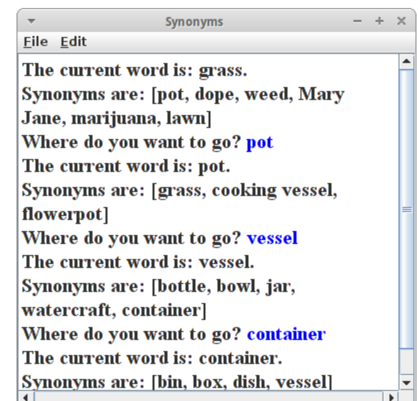
```
Collection<EdgeDirected<String>> outEdgs =

    diGraph.outgoingEdges(currentVertex);
```

die Verbindungsedges auflisten, die von diesem Knoten ausgehen. An den Enden dieser Edges befinden sich die Räume die man darüber erreichen kann:

```
for (EdgeDirected<String> edge : outEdgs) {
    Vertex<String> rm = diGraph.opposite(currentVertex, edge);
    ...
}
```

So kann man dann durch den ganzen Graphen navigieren, also ein Abenteuer erleben.

Synonyms



Wörter die eine gleiche oder ähnliche Bedeutung haben, nennt man *Synonyme*. Im Internet finden sich Listen von Synonymen, z.B. der Website *WordNet* enthält eine Liste mit vielen Synonymen [12-14]. Auch aus Synonymen kann man einen Graphen basteln. Eine Liste von Synonymen sieht z.B. wie folgt aus:

```
grass, pot
pot, cooking vessel
vessel, container
box, container
dish, container
bin, container
pie, dish

plate, dish
```

Aus diesen Synonymen kann man nun einen ungerichteten Graphen erstellen:

```
GraphEdgeList<String, String> graph;
```

durch den man dann wie im Adventure Spiel hindurch navigieren kann. Die Wörter sind die Vertices,

```
Vertex<String> v1 = diGraph.insertVertex(new Vertex<String>("grass"));

Vertex<String> v2 = diGraph.insertVertex(new Vertex<String>("pot"));
```

und die Synonymbeziehung wird über die Edges abgebildet:

```
AbstractEdge<String> e1 = diGraph.insertEdge(v1, v2, "");
```

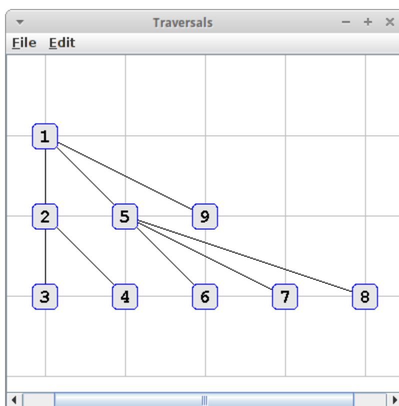
Der Rest vom Code ist fast identisch zum Adventure Beispiel. Weiter unten werden wir noch einen visuellen SynonymBrowser erstellen.

Traversals

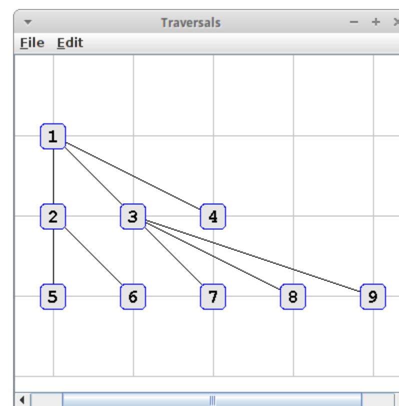
Ein anderes hübsches Beispiel ist es einmal die beiden Traversals für Graphen auszuprobieren und zu visualisieren. Das machen wir mit dem folgende Code:

```
String str = "A{B{C,D},E{F,G,H},I}";
GraphEdgeList<String, String> graph = new GraphParser().parseTree(str);

graph.bfs(new VisitorInterface<Vertex<String>>() {
    int counter = 1;

    public void visit(Vertex<?> p) {
        ((Vertex<String>) p).setElement("" + counter++);
        System.out.println(p);
    }

});
```

In diesem Fall ist der Graph ein Baum, und zufälligerweise der gleiche den wir auch im Kapitel zu den Bäumen verwendet haben. Die Zahl in den Rechtecken gibt die Reihenfolge an in der die jeweiligen Knoten besucht wurden, links für DFS-Traversal und rechts für BFS-Traversal:



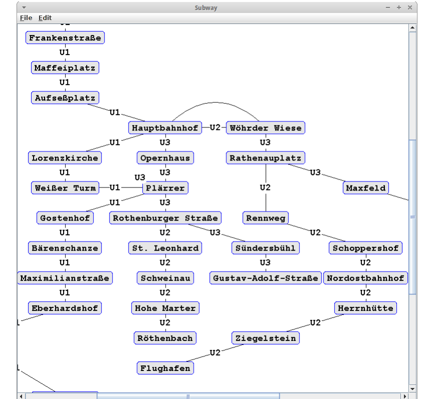
DFS-Traversal



BFS-Traversal

Was wir sehen ist, dass der DFS dem Pre-Order und der BFS dem Level-Order Traversal für Bäume entspricht.

Subway



Ein anderes sehr schönes Beispiel für ein Netzwerk sind UBahnlinien. Z.B. in Nürnberg gibt es drei Linien:

```
U1: Langwasser Süd, Gemeinschaftshaus, Langwasser
Mitte, Scharfreiterring, ...
U2: Röthenbach, Hohe Marter, Schweinau, St.
Leonhard, Rothenburger Straße, ...

U3: Friedrich-Ebert-Platz, Kaulbachplatz, Maxfeld,
Rathenauplatz, ...
```

Auch daraus lässt sich ein Graph erstellen: die Vertices sind die Haltestellen und die Edges stellen die UBahn Linien dar, also U1, U2 oder U3.

Wir lesen wie üblich mit einem BufferedReader Zeile für Zeile und parsen diese in der folgenden Methode:

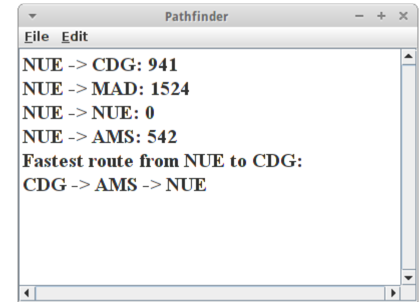
```
private void analyze(String line) {
    StringTokenizer st = new StringTokenizer(line, ",:", false);
    String subwayLine = st.nextToken();

    String from = st.nextToken().trim();
    addVertexToGraph(from);

    while (st.hasMoreTokens()) {
        String to = st.nextToken().trim();
        addVertexToGraph(to);
        graph.insertEdge(vertices.get(from), vertices.get(to),
subwayLine);
        from = to;
    }

}
```

Danach benutzen wir den *GraphDrawerCanvas* um den Graphen ansprechend darzustellen.

```
private void drawRandomGraph(GraphEdgeList<String, String> graph) {
    GraphDrawerCanvas<String, String> canvas = new
GraphDrawerCanvas<String, String>(
            graph);
    canvas.setShapeNode(GraphDrawerCanvas.SHAPE_ROUNDRECT);
    canvas.setOrientation(GraphDrawerCanvas.VERTICAL);
    canvas.setEdgeStyle(GraphDrawerCanvas.EDGE_STYLE_AVOID_OVERLAP);
    canvas.setShowEdgeLabels(GraphDrawerCanvas.EDGE_LABELS_SHOW);
    canvas.setColorVertices(Color.BLUE);
    canvas.setColorVerticesFill(new Color(230, 230, 230));
    canvas.setNodeSeparationX(200);
    canvas.setNodeSeparationY(60);
    canvas.setGridOn(false);
    canvas.setFont("Courier new-bold-18");

 canvas.setTraversalType(GraphDrawingAlgorithm.TRAVERSAL_SPRING_FORCE);
    canvas.execute();
    add(new JScrollPane(canvas), CENTER);

}
```

Weiter unten werden wir das Projekt noch um eine Navigation erweitern.

FlightFinder



Im FlightFinder Projekt wollen wir einen Flug von Nürnberg nach Paris finden. Dazu initialisieren wir erst einmal unseren Graphen,

```
GraphEdgeList<Integer, String> graph = new
GraphEdgeList<Integer, String>();
```

fügen dann die Flughäfen hinzu,

```
Vertex<String> v1 = graph.insertVertex(new
Vertex<String>("NUE"));
Vertex<String> v2 = graph.insertVertex(new Vertex<String>("AMS"));
Vertex<String> v3 = graph.insertVertex(new Vertex<String>("MAD"));

Vertex<String> v4 = graph.insertVertex(new Vertex<String>("CDG"));
```

und schließlich die Verbindungsflüge,

```
AbstractEdge<Integer> e1 = graph.insertEdge(v1, v2, 542);
AbstractEdge<Integer> e2 = graph.insertEdge(v2, v3, 1462);
AbstractEdge<Integer> e3 = graph.insertEdge(v1, v3, 1524);

AbstractEdge<Integer> e4 = graph.insertEdge(v2, v4, 399);
```

Wie wir sehen gibt es keinen direkten Flug von Nürnberg nach Paris. Aber es gibt den Dijkstra Algorithmus, und der findet uns die beste Verbindung. Zunächst initialisieren wir den Algorithmus:

```
Dijkstra<Integer, String> dijK = new Dijkstra<Integer, String>(graph);
```

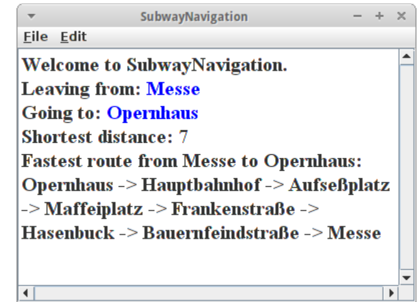
Dann lassen wir uns mal die Distanzen zu allen Flughäfen ausgeben die man von Nürnberg aus direkt und indirekt erreichen kann:

```
Map<Vertex<String>, Integer> dists = dijK.getAllDistances(v1);
for (Vertex<String> vx : dists.keySet()) {
    println(""+v1.getElement()+" -> " + vx.getElement() + ": " +
dists.get(vx));

}
```

Und schließlich würden wir noch unseren genauen Flugplan wissen wollen, wie wir von Nürnberg nach Paris kommen:

```
println("Fastest route from "+v1.getElement()+" to "+v4.getElement()+":
");
Map<Vertex<String>, Vertex<String>> predcrs =
dijK.getAllPredecessors(v1);
Vertex<String> vTmp = v4;
while (vTmp != v1) {
    print(vTmp.getElement() + " -> ");
    vTmp = predcrs.get(vTmp);
}

println(vTmp.getElement());
```

Gute Reise!



SubwayNavigation

Mit den Daten aus dem Projekt Subway können wir auch eine UBahn Navigations Anwendung schreiben. Wir laden wie gehabt die Daten aus der Textdatei in einen Graphen. Allerdings müssen die Elemente der Edges Integer sein, also

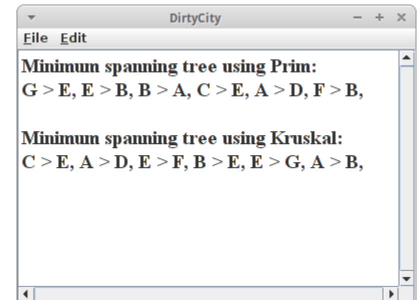
```
GraphEdgeList<Integer, String> graph;
```

sonst ist es nicht möglich die kürzeste Distanz zu berechnen. Idealerweise würde man hier die Fahrtzeiten in Minuten verwenden, da wir aber diese Daten nicht haben, setzen wir einfach alle Distanzen auf 1. Das genügt in einem einfachen Netzwerk, wenn uns nur die Strecke interessiert und nicht die genaue Dauer oder Distanz.

Dann fragen wir den Nutzer nach Abfahrts- und Zielbahnhof:

```
Vertex<String> sourceVertex = getVertex("Leaving from: ");

Vertex<String> destinationVertex  = getVertex("Going to: ");
```

Und dann können wir genau wie beim PathFinder Projekt die Anzahl der Stationen und die Route die zum Ziel führt ausgeben.



DirtyCity

Die Stadt DirtyCity [1] hat uns beauftragt einen Plan auszuarbeiten, wie jedes Haus über eine asphaltierte Straße erreichbar ist, aber gleichzeitig die Kosten für die Asphaltierung möglichst niedrig sein sollen. Wir haben mal ausnahmsweise in der Vorlesung aufgepasst, und da war doch was mit dem Minimum Spanning Tree und den Algorithmen von Prim und Kruskal.

Der Code ist denkbar einfach. Wir übersetzen zunächst die Häuser und ihre Abstände in einen Graphen:

```
GraphEdgeList<Integer, String> graph = new GraphEdgeList<Integer,
String>();

Vertex<String> vA = graph.insertVertex(new Vertex<String>("A"));
Vertex<String> vB = graph.insertVertex(new Vertex<String>("B"));
Vertex<String> vC = graph.insertVertex(new Vertex<String>("C"));
Vertex<String> vD = graph.insertVertex(new Vertex<String>("D"));
Vertex<String> vE = graph.insertVertex(new Vertex<String>("E"));
Vertex<String> vF = graph.insertVertex(new Vertex<String>("F"));
Vertex<String> vG = graph.insertVertex(new Vertex<String>("G"));

AbstractEdge<Integer> e1 = graph.insertEdge(vA, vB, 1);
AbstractEdge<Integer> e2 = graph.insertEdge(vA, vD, 2);
AbstractEdge<Integer> e3 = graph.insertEdge(vB, vC, 3);
AbstractEdge<Integer> e4 = graph.insertEdge(vB, vD, 4);
AbstractEdge<Integer> e5 = graph.insertEdge(vB, vE, 2);
AbstractEdge<Integer> e6 = graph.insertEdge(vB, vF, 2);
AbstractEdge<Integer> e7 = graph.insertEdge(vC, vE, 2);
AbstractEdge<Integer> e8 = graph.insertEdge(vD, vF, 3);
AbstractEdge<Integer> e9 = graph.insertEdge(vE, vF, 2);
AbstractEdge<Integer> e10 = graph.insertEdge(vE, vG, 1);

AbstractEdge<Integer> e11 = graph.insertEdge(vF, vG, 3);
```

wenden dann z.B. den Algorithmus von Prim darauf an:

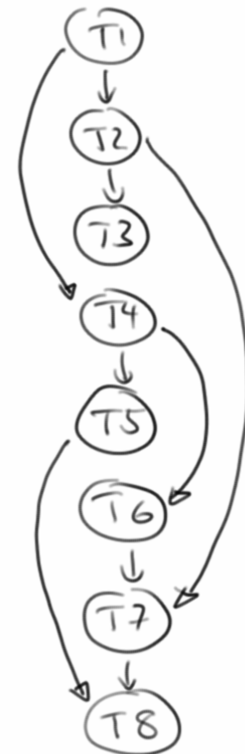
```
Prim<Integer, String> prim = new Prim<Integer, String>(graph);
```

und lassen uns den Minimum Spanning Tree ausgeben:

```
println("Minimum spanning tree using Prim: ");
Map<Vertex<String>, Vertex<String>> tree = prim.findMSP();
for (Vertex<String> vx : tree.keySet()) {
    print(vx.getElement() + " > " + tree.get(vx).getElement() + ", ");

}
```

Jetzt können wir unsere Rechnung stellen. Interessant ist, das Prim einen etwas anderen Minimum Spanning Tree ausgibt als Kruskal. Beide sind aber minimum, bedeutet, der verbaute Asphalt ist der gleiche.

ProjectManagement



Im Projektmanagement möchte man wissen wie lange ein Projekt mindestens dauert. Dazu macht man sich eine Liste von Tasks die zu erledigen sind, schätzt für jeden dieser Tasks ungefähr ab wie lange er dauern wird, und notiert sich Abhängigkeiten, also welche Tasks von anderen Tasks abhängen. Das Resultat einer solchen Schätzung kann man dann in einer Tabelle zusammenfassen:

Task	Duration (days)	Dependencies
T1	5	
T2	5	T1
T3	10	T2
T4	15	T1
T5	5	T4
T6	10	T4
T7	15	T2, T6
T8	10	T5, T7

Was für das weitere Projekt jetzt wichtig wäre ist zu wissen in welcher Reihenfolge die Tasks abgearbeitet werden sollen und den kritischen Pfad [15] zu identifizieren. Über den kritischen Pfad kann man dann auch die Mindestlaufzeit des Projektes abschätzen.

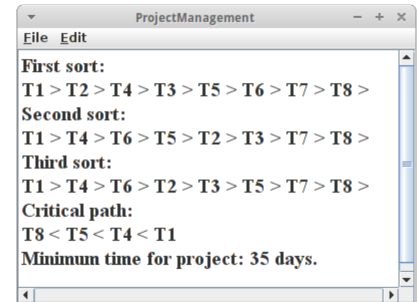
Für den ersten Schritt machen wir aus unsere Taskliste einen DAG, also einen azyklischen Digraphen. Azyklisch deshalb, denn würde die Taskliste eine Kreis enthalten, z.B. T1 hängt von T2 ab und T2 wiederum von T1, dann würde das Projekt nie fertig werden.

```
DiGraphEdgeList<Integer, String> graph = new DiGraphEdgeList<Integer,
String>();

Vertex<String> v1 = graph.insertVertex(new Vertex<String>("T1"));
Vertex<String> v2 = graph.insertVertex(new Vertex<String>("T2"));
Vertex<String> v3 = graph.insertVertex(new Vertex<String>("T3"));
Vertex<String> v4 = graph.insertVertex(new Vertex<String>("T4"));
Vertex<String> v5 = graph.insertVertex(new Vertex<String>("T5"));
Vertex<String> v6 = graph.insertVertex(new Vertex<String>("T6"));
Vertex<String> v7 = graph.insertVertex(new Vertex<String>("T7"));
Vertex<String> v8 = graph.insertVertex(new Vertex<String>("T8"));
```

```
AbstractEdge<Integer> e1 = graph.insertEdge(v1, v2, 1);
AbstractEdge<Integer> e2 = graph.insertEdge(v1, v4, 1);
AbstractEdge<Integer> e3 = graph.insertEdge(v2, v7, 1);
AbstractEdge<Integer> e4 = graph.insertEdge(v2, v3, 1);
AbstractEdge<Integer> e5 = graph.insertEdge(v4, v5, 1);
AbstractEdge<Integer> e6 = graph.insertEdge(v4, v6, 1);
AbstractEdge<Integer> e7 = graph.insertEdge(v5, v8, 1);
AbstractEdge<Integer> e8 = graph.insertEdge(v6, v7, 1);
AbstractEdge<Integer> e9 = graph.insertEdge(v7, v8, 1);
```

Interessant ist hier vielleicht, dass wir die Dauer der Tasks noch gar nicht verwenden. Das hat damit zu tun, dass die Dauer zu den Vertices gehört, nicht den Edges.

Die Reihenfolge in der die Tasks abgearbeitet werden sollen ermitteln wir einfach via der topologischen Sortierung:




```
TopologicalSort<Integer, String> toso =
    new TopologicalSort<Integer, String>(graph);
println("First sort: ");
Collection<Vertex<String>> verts = toso.sort();
for (Vertex<String> vertex : verts) {
    print(vertex.getElement() + " > ");

}
```

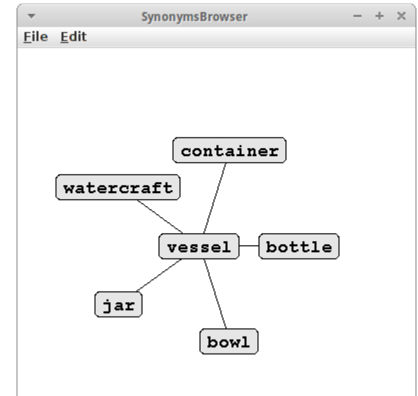
Die topologischen Sortierung ist nicht ganz eindeutig, es gibt nicht selten mehrere mögliche Antworten. Das Resultat in unserem Beispiel besagt, dass wir mit T1 beginnen sollten, und dass T8 der letzte Task ist. Das ist schon mal nicht schlecht, es sagt uns grob die Reihenfolge in der die Tasks abgearbeitet werden sollten. Allerdings könnten einige Tasks auch parallel abgearbeitet werden.

Um herauszufinden welche Tasks nicht parallel abgearbeitet werden können, benötigen wir den kritischen Pfad. Hier hilft uns unser alter Freund, der Dijkstra Algorithmus. Wir geben ihm den Anfangs- und Endtask die wir aus der topologischen Sortierung erhalten haben, und lassen uns dann den kürzesten Pfad zwischen den beiden geben:

```
Dijkstra<Integer, String> dijK =
    new Dijkstra<Integer, String>(graph);

Map<Vertex<String>, Vertex<String>> predcrs =
    dijK.getAllPredecessors(startVertex);
Vertex<String> vTmp = endVertex;
while (vTmp != startVertex) {
    print(vTmp.getElement() + " < ");
    vTmp = predcrs.get(vTmp);
}

println(vTmp.getElement());
```

Das ist der kürzeste Pfad. Wenn wir jetzt einfach aufaddieren wie lange jeder dieser Tasks benötigt wissen wir auch wie lange unser Projekt mindestens dauern wird. Es kann aber natürlich auch länger dauern!

Challenges



SynonymsBrowser

Im Beispiel SynonymsBrowser erfolgt die Navigation durch den Synonymgraphen auf visuelle Art und Weise. Dazu stellt man in der Mitte das Ursprungswort dar, und zeichnet im Kreis darum die Synonyme. Der Nutzer kann dann auf die Synonyme klicken, um dann von einem Wort zum nächsten zu gelangen.

Die Vorarbeit haben wir schon im Projekt *Synonyms* geleistet, d.h. den Graphen haben wir schon. Wir wollen aber nicht den ganzen Graphen anzeigen, sondern lediglich einen Teil davon. Den besorgen wir uns über die Methode *getSubGraph()*, die ausgehend von einem Startvertex nach den nächsten Nachbarn sucht, und daraus einen kleinen Graphen macht:

```
private GraphEdgeList<Integer, String> getSubGraph(String startWord) {
    GraphEdgeList<Integer, String> subGraph = new GraphEdgeList<Integer,
String>();

    Vertex<String> startVertex = graph.findVertex(startWord);
    subGraph.insertVertex(startVertex);
    List<Vertex<String>> wordsToGo = findWordsToGoTo(startVertex);
    for (Vertex<String> vertex : wordsToGo) {
        if (!subGraph.containsVertex(vertex)) {
            subGraph.insertVertex(vertex);
        }
        subGraph.insertEdge(startVertex, vertex, null);
    }
    return subGraph;

}
```

Diesen kleinen Graphen zeichnen wir dann mit der Methode *drawGraph()*:

```
private void drawGraph(GraphEdgeList<Integer, String> subGraph) {
    GraphDrawingAlgorithm<Integer, String> algorithm =
        new GraphDrawingAlgorithmBFSRadialMostImportant<Integer, String>(
            subGraph, EDGE_LENGTH);
    algorithm.execute();

    removeAll();

    // drawNodes:
    Collection<Vertex<String>> vertices = subGraph.vertices();
    for (Vertex<String> vertex : vertices) {
        HashMap<Vertex<String>, Point> positions =
            algorithm.getPositions();
        Point pt = positions.get(vertex);
        String tmp = vertex.getElement().toString();
        drawElement(tmp, pt.x, pt.y);
    }

    // drawEdges:
    Collection<AbstractEdge<Integer>> edges = subGraph.edges();
    for (AbstractEdge<Integer> edge : edges) {
```

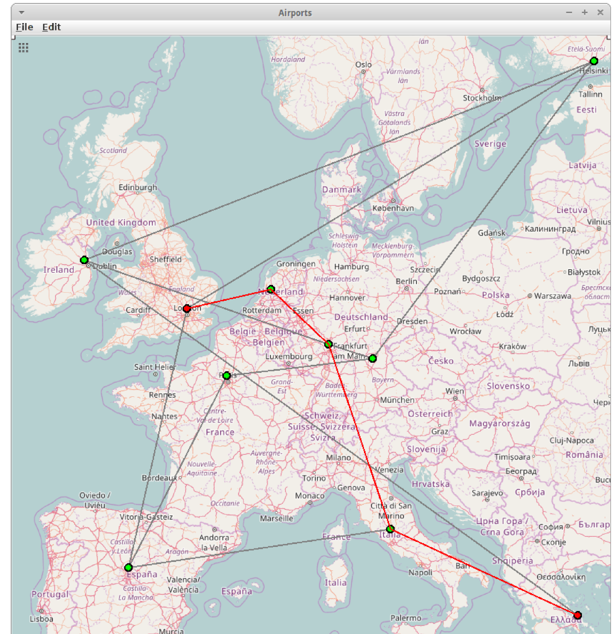
```
            Vertex<String>[] vtcs = (Vertex<String>[]) edge.getVertices();
            HashMap<Vertex<String>, Point> positions =
                algorithm.getPositions();
            drawEdge(positions.get(vtcs[0]), positions.get(vtcs[1]));
        }
    }
```

dabei benutzen wir einfach den *GraphDrawingAlgorithmBFSRadialMostImportant* Algorithmus der die Positionen der Vertices berechnet. Die Methode *drawEdge()* tut genau das was sie sagt. Was noch fehlt ist der MouseListener, das wird in der Methode *drawElement()* erledigt:

```
    private void drawElement(String element, int x, int y) {
        GLabel nodeLbl = new GLabel(element.trim());
        nodeLbl.setFont("Courier new-bold-18");
        nodeLbl.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                // System.out.println(((GLabel)e.getSource()).getLabel());
                reDrawGraph(((GLabel) e.getSource()).getLabel());
            }
        });
        double lblW = nodeLbl.getWidth();
        double lblH = nodeLbl.getHeight();
        double lblA = nodeLbl.getAscent();

        int dx = ((int) lblW + 2 * 7);
        int dy = ((int) lblH + 2 * 2);

        GRoundRect node = new GRoundRect(dx, dy);
        node.setFilled(true);
        node.setFillColor(new Color(230, 230, 230));

        add(node, x - dx / 2 - 1 + OFFSET_X, y - dy / 2 + OFFSET_Y);
        add(nodeLbl, x - lblW / 2 + OFFSET_X, y + lblA / 2 - 1 + OFFSET_Y);
    }
```

Die Methode *reDrawGraph(String word)* zeichnet den ganzen Graph neu, jetzt aber beginnend mit einem neuen Wort.

Airports



In diesem Projekt wollen wir eine graphische Version von FlightFinder implementieren. Im ersten Schritt müssen wir uns eine Karte für die Region besorgen die uns interessiert, da ist z.B. OpenStreetMaps ganz nützlich. Hier müssen wir uns natürlich die Längen- und Breitengrade merken. Dann benötigen wir Daten zu den Flughäfen, also den IATA Airport Code, ihre Position, sowie die Distanzen zwischen den Flughäfen, findet man alles im Internet. Das Ganze speichert man am besten in eine Textdatei (oder eine Datenbank):

```
# size of map
61, 36,-11, 26
# location: latitude, longitude (data
is only approx. correct)
FRA, 49.3,  8.4
NUE, 48.7, 11.1
HEL, 61.1, 24.8
LHR, 50.8, -0.37
AMS, 51.6,  4.8
DUB, 52.8, -6.7
MAD, 40.0, -4.0
CDG, 48.0,  2.1
FCO, 41.6, 12.2
ATH, 38.0, 23.8
# distances
    , LHR, CDG, AMS, FRA, MAD, FCO, DUB, HEL, ATH, NUE
LHR,   0,    , 371,    ,1247,    ,    ,1847,    ,
CDG,    ,   0,    ,    ,1065,    ,    ,    ,    , 620
AMS,    ,    ,   0, 365,    ,    ,    ,    ,    ,
FRA,    ,    ,    ,   0,    , 960,1087,    ,    , 190
MAD,    ,    ,    ,    ,   0,1332,    ,    ,    ,
FCO,    ,    ,    ,    ,    ,   0,    ,    ,1086,
DUB,    ,    ,    ,    ,    ,    ,   0,2023,2877,
HEL,    ,    ,    ,    ,    ,    ,    ,   0,    ,1489
ATH,    ,    ,    ,    ,    ,    ,    ,    ,   0,

NUE,    ,    ,    ,    ,    ,    ,    ,    ,    ,   0
```

Als erstes lesen wir die Ausmaße unserer Map ein:

```
private double latMin = 0;
private double latMax = 0;
private double lonMin = 0;
private double lonMax = 0;

private void getBoundaries(String line) {
    String[] data = line.split(",");
    latMax = Double.parseDouble(data[0].trim());
    latMin = Double.parseDouble(data[1].trim());
    lonMin = Double.parseDouble(data[2].trim());
    lonMax = Double.parseDouble(data[3].trim());

}
```

Danach merken wir uns für später die Positionen unserer Flughäfen:

```
private Map<String, Location> airportLocations = new HashMap<String,
Location>();

private void addToCities(String line) {
    String[] data = line.split(",");
    Location loc = new Location(Double.parseDouble(data[1].trim()),
            Double.parseDouble(data[2].trim()));
    airportLocations.put(data[0].trim(), loc);

}
```

Hier haben wir die kleine, private Hilfsklasse *Location* eingeführt, die ganz nützlich ist, man hätte aber auch die AWT-Klasse *Point* verwenden können:

```
private class Location {
    public double latitude;
    public double longitude;

    public Location(double latitude, double longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }

}
```

Nach diesen Vorarbeiten müssen wir unseren Graphen konstruieren. Die Flughäfen werden die Vertices und die Verbindungsstrecken werden die Edges, mit den Entfernungen als Element:

```
private GraphEdgeList<Integer, String> graph;
private Map<String, Vertex<String>> vertices;
private boolean firstDistanceLine = true;
private String[] airportKeys = null;

private void addToDistances(String line) {
    if (firstDistanceLine) {
        airportKeys = line.split(",");
        for (int i = 0; i < airportKeys.length; i++) {
            airportKeys[i] = airportKeys[i].trim();
        }
        firstDistanceLine = false;
    } else {
        String[] data = line.split(",");
        String from = data[0].trim();
        for (int i = 1; i < data.length; i++) {
            String to = airportKeys[i];
            if (data[i].trim().length() > 0) {
                int distance = Integer.parseInt(data[i].trim());
                if (distance > 0) {
                    if (!vertices.containsKey(from)) {
                        Vertex<String> vtx = graph
                                .insertVertex(new Vertex<String>(from));
                        vertices.put(from, vtx);
                    }
                    if (!vertices.containsKey(to)) {
                        Vertex<String> vtx = graph
                                .insertVertex(new Vertex<String>(to));
                        vertices.put(to, vtx);
                    }
```

```
                        graph.insertEdge(vertices.get(from),
                            vertices.get(to), distance);
                    }
                }
            }
        }

    }
```

Im nächsten Schritt müssen wir uns um die GUI kümmern. Wir laden als erstes die Karte als Bild und fügen sie unserem Canvas hinzu:

```
    private void loadMap(String fileName) {
        GImage map = new GImage(fileName);
        this.setSize((int) map.getWidth(), (int) map.getHeight());
        add(map);

    }
```

Dann zeichnen wir die Flughäfen als GOvals ein:

```
    private Map<GObject, String> airportNames = new HashMap<GObject,
    String>();

    private void displayAirports() {
        for (String city : airportLocations.keySet()) {
            Location loc = airportLocations.get(city);
            Point p = scalePoint(loc);
            GOval airport = new GOval(p.x - RADIUS, p.y - RADIUS, RADIUS * 2,
                    RADIUS * 2);
            airport.setFilled(true);
            airport.setFillColor(Color.GREEN);
            airport.addMouseListener(new MouseAdapter() {
                ...
            });
            add(airport);
            airportNames.put(airport, city);
        }


    }
```

Die Map *airportNames* benötigen wir, wenn wir später auf Mouseklicks reagieren wollen. Denn beim Mouseklick bekommen wir eine Referenz auf den GOval, der weiß aber nicht was sein Name ist. Deswegen brauchen wir die Map. Die Methode *scalePoint()* sorgt dafür, dass die Flughäfen an die richtige Stelle skaliert werden:

```
    private Point scalePoint(Location loc) {
        double scaleY = getHeight() / (latMax - latMin);
        double scaleX = getWidth() / (lonMax - lonMin);
        int y = (int) (getHeight() - (loc.latitude - latMin) * scaleY + 40);
        int x = (int) ((loc.longitude - lonMin) * scaleX + 5);
        Point p = new Point(x, y);
        return p;

    }
```

Jetzt sehen wir also die Map und die Flughäfen. Es fehlen noch die Verbindungsstrecken:

```
private void displayConnections() {
    Collection<AbstractEdge<Integer>> edges = graph.edges();
    for (AbstractEdge<Integer> edge : edges) {
        int distance = edge.getElement();
        Vertex<?>[] vtcs = edge.getVertices();
        Vertex<String> from = (Vertex<String>) vtcs[0];
        Vertex<String> to = (Vertex<String>) vtcs[1];

        Location locFrom = airportLocations.get(from.getElement());
        Location locTo = airportLocations.get(to.getElement());
        drawLine(locFrom, locTo, Color.GRAY);
    }
}
```

Jetzt steht die UI und es fehlt lediglich die Interaktion, also die Mouseklicks. Dafür haben wir in der Methode *displayAirports()* aber schon Vorkehrungen getroffen, wir müssen lediglich das folgende ergänzen:

```
private int clickCount = 0;
private String departure;
private String destination;

private void displayAirports() {
    ...
        airport.addMouseListener(new MouseAdapter() {
            @Override
            public void mousePressed(MouseEvent e) {
                ((GOval) e.getSource()).setFillColor(Color.RED);
                if (clickCount % 2 == 0) { // first
                    departure =
                        airportNames.get(((GOval) e.getSource()));
                } else { // second
                    destination =
                        airportNames.get(((GOval) e.getSource()));
                    findBestRoute(departure, destination);
                }
                clickCount++;
            }
        });
    ...
}
```

In der Methode *findBestRoute()* rufen wir wie im Projekt FlightFinder einfach den Dijkstra Algorithmus auf, der uns dann auf der Konsole die kürzeste Distanz und die Strecke ausgibt:

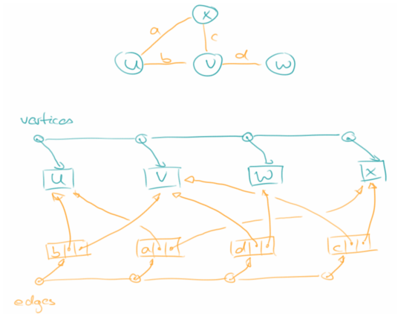
```
Shortest distance from ATH to LHR: 2782

LHR < AMS < FRA < FCO < ATH
```

Man kann die Strecke natürlich auch noch visuell markieren.

Graph Datastructures

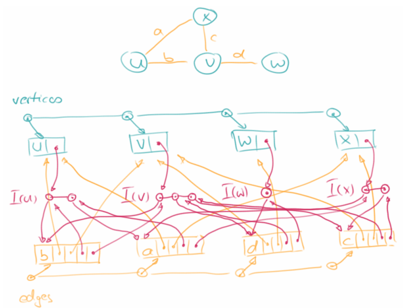
In allen Beispielen die wir bisher verwendet haben, war die zugrundliegende Datenstruktur für die Graphen immer eine sogenannte *EdgeList*. Das hat bisher ganz gut funktioniert. Ähnlich wie bei ArrayList und LinkedList, gibt es aber auch für Graphen unterschiedliche Datenstrukturen, mit unterschiedlichen Eigenschaften. Wir werden diese im Folgenden kurz darlegen, und die Idee hinter dieser Übung ist, diese dann auch zu implementieren, weitere Details finden sich in [2].

Edge List



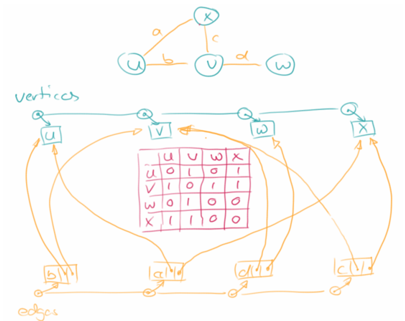
Die Edge List ist die einfachste Art einen Graphen darzustellen. Sie besteht aus zwei Listen (oder Sets), eine für die Vertices und eine für die Edges. Die Vertices sind ganz einfach und enthalten lediglich Information zu ihren Elementen. Die Edges haben zum einen Platz für ihre Elemente, aber sie haben auch je eine Referenz auf die beiden Vertices die sie miteinander verbinden. Die Edge List benötigt wenig Platz, O(n+m), wobei n die Anzahl der Vertices und m die Anzahl der Edges ist. Diese Datenstruktur ist einfach zu implementieren und von der Geschwindigkeit her durchaus akzeptabel.

Adjacency List



Die Adjacency List ist eine weitere Art einen Graphen darzustellen. Sie besteht auch aus zwei Listen, je eine für die Vertices und eine für die Edges. Zusätzlich gibt es aber noch Adjacency-Objekte: diese erlauben es z.B. sehr schnell die Edges die zu einem Vertex gehören zu finden, und damit dann die Vertices zu finden die benachbart, also adjacent sind. Sie ist etwas komplizierter zu implementieren, ist aber dafür für die meisten Anforderungen die effizienteste. Die Platzanforderungen liegen auch bei O(n+m).

Adjacency Matrix



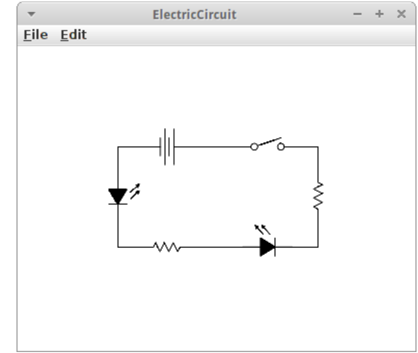
Die Adjacency Matrix ist eigentlich die natürlichste Art einen Graphen darzustellen, und war auch die erste. Sie besteht auch aus zwei Listen, je eine für die Vertices und eine für die Edges. Ausserdem gibt es aber noch eine Matrix, eben besagte Adjacency Matrix, die Nullen enthält falls zwei Vertices nicht adjacent sind, und Einsen falls schon. Sie ist einfach zu implementieren und besonders nützlich wenn man die Methode *areAdjacent()* häufig benötigt. Ein Nachteil ist allerdings, dass sie viel Platz braucht, O(n^2), und für Schreiboperationen sehr langsam ist.

Asymptotic Performance

Die folgende Tabelle [2] gibt einen Überblick über die Performanz der verschiedenen Implementierungen, was einem bei der Wahl behilflich sein kann.

	Edge List	**Adjacency List**	**Adjacency Matrix**
Space	n+m	n+m	n^2
incidentEdges()	m	deg(v)	n
areAdjacent()	m	min(deg(v), deg(w))	1
insertVertex()	1	1	n^2
insertEdge()	1	1	1
removeVertex()	m	deg(v)	n^2
removeEdge()	1	1	1

ElectricCircuit



Elektrische Schaltkreise können auch durch Graphen dargestellt werden. Z.B., könnte man einen Schaltkreis wie folgt beschreiben:

```
String edges = "EC_Battery-EC_LED-EC_Resistor-
    EC_LED-EC_Resistor-EC_Switch";
```

Im Fall des elektrischen Schaltkreises sind es die Edges die alle Informationen enthalten, in diesem Fall welches Bauelement an welche Stelle kommt. Dabei ist aber die genaue Position egal, es kommt lediglich auf die relative Position an. Auch ein schönes Beispiel wie vielseitig unsere Graph Klassen sind.

Fragen

1. Was ist der Unterschied zwischen Dijkstras Algorithmus und Kruskals Algorithmus?

2. Hat der folgende Pfad Cyclen? Mit welchem Algorithmus findet man Cyclen?

   ```
   P1 = (U,f,X,g,V,b,W)
   ```

3. Zeichnen Sie den Graphen, der sich aus der nachfolgenden Nachbarschaftsmatrix ergibt.

	A	B	C	D	E
A	0	3	infty	infty	infty
B		0	2	infty	6
C			0	4	3
D				0	3
E					0

4. Finden Sie den kürzesten Weg von A nach E in der obigen Grafik, und erklären Sie den Algorithmus den Sie verwenden.

5. Für die folgenden Beispiele identifizieren Sie die Ecken und die Kanten.
   - Beziehungen zwischen Forschern, z.B. über gemeinsame Papers oder Bücher
   - Vererbung zwischen Klassen in einer objektorientierten Sprache
   - Elektrische Verdrahtung oder Wasserleitungen
   - Einen Stadtplan
   - Das Internet

6. Alice liebt Fremdsprachen und will ihren Vorlesungsplan für die nächsten Jahre festlegen. Sie interessiert sich für die folgenden Kurse: L15, L16, L22, L31, L32, L169 und L141. Es gibt allerdings gewisse Abhängigkeiten zwischen den Vorlesungen. Finden Sie die Reihenfolge der Kurse, die Alice erlaubt, alle Voraussetzungen zu erfüllen (verwenden Sie einen Graphen).
   - L15: L16
   - L16: (none)
   - L22: L15
   - L31: L16
   - L32: (none)
   - L169: L22
   - L141: L31, L32

Referenzen

Es werden wieder weniger Referenzen, bedeutet wohl, dass wir langsam zum Ende des Buches kommen.

[1] Computer Science Unplugged, http://csunplugged.com/

[2] Data Structures and Algorithms in Java, M.T. Goodrich and R. Tamassia

[3] Edsger W. Dijkstra, https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

[4] Dijkstra's algorithm, https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[5] A* search algorithm, https://en.wikipedia.org/wiki/A*_search_algorithm

[6] Prim's algorithm, https://en.wikipedia.org/wiki/Prim%27s_algorithm

[7] Algorithmus von Prim, https://de.wikipedia.org/wiki/Algorithmus_von_Prim

[8] Kruskal's algorithm, https://en.wikipedia.org/wiki/Kruskal's_algorithm

[9] Disjoint-set data structure, https://en.wikipedia.org/wiki/Disjoint-set_data_structure

[10] Directed acyclic graph, https://en.wikipedia.org/wiki/Directed_acyclic_graph

[11] Topological sorting, https://en.wikipedia.org/wiki/Topological_sorting

[12] What is WordNet?, https://wordnet.princeton.edu

[13] George A. Miller (1995). WordNet: A Lexical Database for English.
Communications of the ACM Vol. 38, No. 11: 39-41.

[14] Christiane Fellbaum (1998, ed.) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.

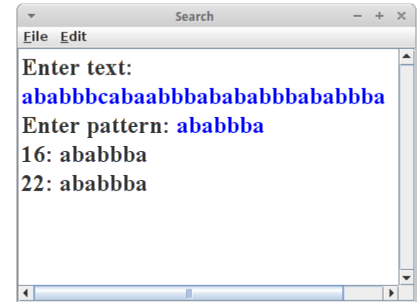
[15] Critical path method, https://en.wikipedia.org/wiki/Critical_path_method

[16] OpenStreetMap, https://www.openstreetmap.org/

Text

Die wichtigsten Anwendungen für die wir unsere Computer verwenden haben mit Text zu tun. Das beginnt wahrscheinlich mit dem Browser, über Email und geht bis hin zu unserem Lieblings-Textverarbeitungsprogramm. Selbst SMS und WhatsApp haben mit Text zu tun. Etwas das wir dabei als sebstverständlich voraussetzen ist die *Suche*: sowohl in unseren Word Dokumenten, in unseren Dateien, und auch im Web. Ein Teil dieses Kapitel widmet sich effektiver Such-Algorithmen. Dazu gehören aber auch Spellchecker, Phonetische Suche und z.B. Plagiatserkennung. Dokumente können dabei sowohl Text Dokumente, wie z.B. eine Bachelorarbeit, aber auch HTML Dokumente sein. Aber auch eine DNA Sequenz kann man als Dokument bezeichnen, und unter bestimmten Umständen macht es auch Sinn digitalisierte Bilder als Dokumente zu bezeichnen. Viele der Algorithmen die wir hier sehen, werden u.a. in Spellcheckern, in DNA Analyse und auch in Spam Filtern verwendet, aber auch in der Betrugserkennung und der Fingerabdruckanalyse.

Search



Sehr häufig wollen wir ein gewisses Wort in einem Text finden. Dann verwenden wir einen Suchalgorithmus. Wir werden uns hier zwei näher ansehen. Der erste ist einfach ein Brute-Force Algorithmus: wir suchen den Pattern *pat* in dem Text *txt*, und die Methode soll ab der Position *shift* anfangen zu suchen:

```
private int search(String txt, String pat, int
shift) {
    final int m = pat.length();
    final int n = txt.length();

    for (int i = shift; i <= (n - m); i++) {
        if (txt.substring(i, i + m ).equals(pat)) {
            return i;
        }
    }

    return -1;

}
```

Der Algorithmus geht im String *txt* einfach eine Position nach der anderen durch, beginnend bei *shift*, und vergleicht ob der Substring mit der Länge des Patterns ab der Position i im Text, gleich dem gesuchten Pattern ist. Wenn ja gibt er die Position zurück, ansonsten versucht er es weiter.

Der Brute-Force Algorithmus ist nicht der schnellste, sein Laufzeitverhalten ist im Schnitt O(m*n).

BoyerMoore

Etwas besser macht das der *Boyer-Moore* Algorithmus. Den versteht man am besten wenn man sich gleich ein konkretes Beispiel ansieht: wir versuchen in dem Text,

```
HERE IS A SIMPLE EXAMPLE
```

den Pattern,

```
EXAMPLE
```

zu finden. Wir beginnen damit, dass wir die beiden unter einander schreiben,

```
HERE IS A SIMPLE EXAMPLE

EXAMPLE
```

Wir vergleichen den letzten Buchstaben im Pattern, 'E', mit dem Buchstaben an der gleichen Stelle im Text, dem 'S'. Da die beiden nicht gleich sind, brauchen wir die anderen Buchstaben davor gar nicht zu vergleichen.

Interessant ist aber weiter, dass das 'S' gar nicht im Pattern vorkommt. Deswegen dürfen wir für den nächsten Vergleich, den Pattern um seine ganze Länge nach rechts verschieben,

```
HERE IS A SIMPLE EXAMPLE

       EXAMPLE
```

Jetzt vergleichen wir das 'E' mit dem 'P'. Auch hier kein Match. Allerdings kommt das 'P' dieses mal im Pattern vor, und zwar an vor-vorletzter Stelle. Die Regel besagt jetzt, dass wir den Pattern so verschieben müssen, dass diese beiden 'P's untereinander sind:

```
HERE IS A SIMPLE EXAMPLE
          EXAMPLE
```

Nach diesem Verschieben, beginnen wir wieder die beiden von rechts nach links zu vergleichen:

```
HERE IS A SIMPLE EXAMPLE
          EXAMPLE
```

Das geht ganz gut aber beim 'A' gibt es keinen Match mehr. Also wieder nach dem 'I' im Pattern suchen. Kommt nicht vor, also dürfen wir den Pattern bis nach dem 'I' nach rechts verschieben:

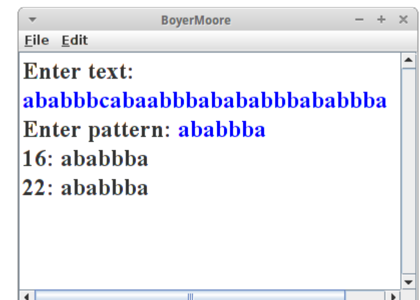
```
HERE IS A SIMPLE EXAMPLE
             EXAMPLE
```

Jetzt müssen wir nach dem 'X' im Pattern suchen, und den Pattern um soviel verschieben, dass das 'X' unter dem 'X' ist:

```
HERE IS A SIMPLE EXAMPLE
                 EXAMPLE
```

Wenn wir nun wieder unseren Vergleich von rechts beginnend machen, dann haben wir unseren Match gefunden.

Was wir hier verwendet haben ist die *Bad-Character Regel* des *Boyer-Moore* Algorithmus. Es gibt aber noch die *Good-Suffix Regel*, um die zu verstehen schaut man sich aber das Beispiel beim Herrn Moore persönlich an [1]. Das Laufzeitverhalten von Boyer-Moore ist im schlimmsten Fall zwar auch O(m*n), aber wenigstens für englischen Text hat er meistens ein Laufzeitverhalten von O(m+n+x), also praktisch linear in m+n, und das ist gut.



In Code sieht das dann so aus:

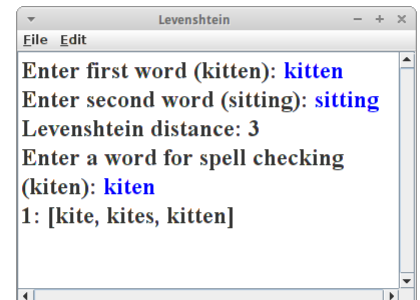
```
private int search(String txt, String pat, int
shift) {
    final int m = pat.length();
    final int n = txt.length();

    int[] badchar = new int[NO_OF_CHARS];
    initBadCharHeuristic(pat, m, badchar);

    while (shift <= (n - m)) {
        int j = m - 1;

        // chars of pattern and text are matching
        while (j >= 0 && pat.charAt(j) == txt.charAt(shift + j)) {
            j--;
        }

        if (j < 0) {
            return shift; // we found pattern
        } else {
            shift += Math.max(1, j - badchar[txt.charAt(shift + j)]);
        }
    }
    return -1;

}
```

wobei,

```
private void initBadCharHeuristic(String str, int size, int[] badchar) {
    // initialize all occurrences as -1
    for (int i = 0; i < NO_OF_CHARS; i++) {
        badchar[i] = -1;
    }

    // fill the actual value of last occurrence of a character
    for (int i = 0; i < size; i++) {
        badchar[str.charAt(i)] = i;
    }
}
```

Es gibt noch viele andere Pattern-Matching Algorithmen. Bekannt sind er Rabin-Karp Algorithmus [2] und der Knuth Morris Pratt (KMP) Algorithmus [3], welcher vor allem deswegen interessant ist weil er ein Laufzeitverhalten von O(m+n) hat.

Levenshtein



Wie funktioniert denn ein Spellchecker? Ein Spellchecker versucht Wörter zu finden, die ähnlich sind wie das Wort das wir falsch geschrieben haben. Was bedeutet ähnlich? Es heißt soviel wie, dass es in der Nähe ist, bzw. eine kleine Distanz entfernt ist. Wie misst man jetzt Distanzen zwischen zwei Strings? Dafür gibt es sogenannte String-Metriken [4] und die bekannteste dürfte die *Levenshtein* Distanz sein [5]. Man kann sie auch zur Korrektur bei der Digitalisierung von Dokumenten (OCR) verwenden, auch in der Genetik werden String-Metriken und ihre Nachfahren eingesetzt.

Die *Levenshtein* Distanz ist eine sogenannte Editierdistanz, d.h. sie zählt wieviele Edits minimal nötig sind um von einem String auf den anderen zu kommen. Edits sind dabei Einfüge-, Lösch- und Ersetz-Operationen. Als Beispiel berechnen wir die *Levenshtein* Distanz zwischen den Worten "kitten" und "sitting". Es sind drei Edits notwendig:

- ersetze 'k' durch 's',
- ersetze 'e' durch 'i' und
- füge ein 'g' am Ende an.

deswegen ist die *Levenshtein* Distanz 3.

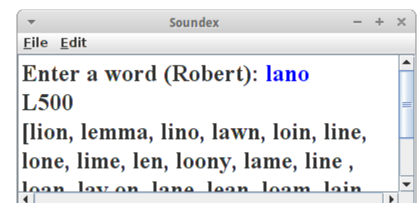
Der Algorithmus ist ein rekursiver:

```
private int distance(String s, String t) {
    int cost;
    int len_s = s.length();
    int len_t = t.length();

    // base case:
    if (len_s == 0 || len_t == 0) {
        return len_s + len_t;
    }

    // test if last characters of the strings match
    if (s.charAt(len_s - 1) == t.charAt(len_t - 1)) {
        cost = 0;
    } else {
        cost = 1;
    }
```

```
        // return minimum of delete char from s, delete char from t, and
        // delete char from both
        return min(
                distance(s.substring(0, len_s - 1), t) + 1,
                distance(s, t.substring(0, len_t - 1)) + 1,
                distance(s.substring(0, len_s - 1), t.substring(0, len_t -
    1))
                        + cost);

    }
```

Diese Version ist nicht die schnellste, aber die verständlichste [5].

Interessant in diesem Zusammenhang ist auch noch die *Hamming* Distanz [6]: sie funktioniert nur für Strings gleicher Länge und entspricht einfach der Anzahl der Buchstaben die unterschiedlich sind. Etwas später werden wir noch die Longest Common Subsequence sehen. Die Referenz [7] ist eine interessante Quelle in diesem Zusammenhang.



Soundex

Beim Soundex Algorithmus [8] geht es darum Homophone, also gleichklingende Worte, zu finden. Z.B. "Robert" und "Rupert" klingen im Englischen sehr ähnlich, obwohl sie unterschiedlich geschrieben werden. Verwendet wird der Soundex z.B. für die phonetische Suche.

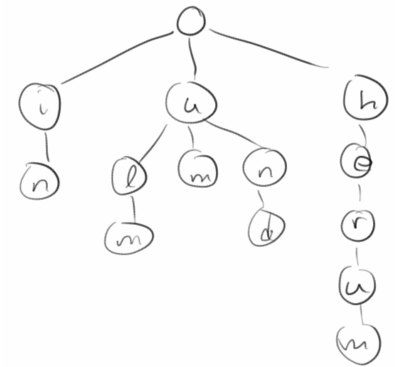
Der Soundex Algorithmus macht aus jedem Wort einen Soundex Code der aus einem Buchstaben (dem Anfangsbuchstaben) gefolgt von 3 Ziffern besteht. Z.B. wird aus "Robert" und "Rupert" der Soundex Code "R163". Berechnet wird der Soundex Code wie folgt:

1. zunächst wandelt man den String in Großbuchstaben um;
2. dann merkt man sich den ersten Buchstaben;
3. vom String werden all "H" und "W" entfernt, außer dem ersten Buchstaben;
4. alle Konsonanten werden durch Zahlen ersetzt, und zwar: "BFPV" -> 1, "CGJKQSXZ" -> 2, "DT" -> 3, "L" -> 4, "MN" -> 5, "R" -> 6;
5. es sollen keine Ziffern doppelt vorkommen, also z.B. aus "77757" wird "757";
6. die Vokale "AEIOUY" (außer der erste Buchstabe) werden entfernt;
7. zum Schluß wird der erste Buchstabe Buchstabe in dem String wieder mit dem ursprünglichen ersten Buchstaben ersetzt, dann werden drei Nullen angehängt, und davon dann die ersten vier Zeichen genommen. Das ist der Soundex Code.

Beispiele sind: "Robert" und "Rupert": R163, "Rubin": R150, "Ashcraft" und "Ashcroft": A261, "Tymczak": T522 und "Pfister": P236. Fertige Algorithmen findet man u.a. in der Apache Commons Codec [9].

Der Soundex Algorithmus ist für die englische Sprache optimiert, es gibt aber auch eine Version fürs Deutsche [10]. Der Algorithmus wurde ursprünglich von Robert C. Russell and Margaret King Odell entwickelt und patentiert. Heutzutage wird eher der *Metaphone* Algorithmus verwendet, der auch für andere Sprachen sehr gut funktioniert. Eine schöner Vergleich findet sich in [11].

Tries



Kommen wir zu einer ganz interessanten Datenstruktur, dem *Trie*. Das Wort ist eine Anspielung auf Tree, denn es handelt sich um eine Baumdatenstruktur, aber kommt von dem Wort *re**trie**val*, denn dafür wurde sie erfunden: zum Suchen. Ein *Trie* ist eine Datenstruktur in der man z.B. alle Wörter eines Dokumentes speichern kann. Er erlaubt es einem dann sehr schnell zu suchen, insbesondere auch nach Prefixen und Pattern [12,13].

Da es relativ aufwendig ist einen *Trie* anzulegen, macht es nur Sinn mit Tries zu arbeiten, wenn man häufiger als nur einmal ein Suche in einem Dokument machen muss. Deswegen eignen sie sich z.B. hervorragend für Suchmaschinen.

Standard Tries

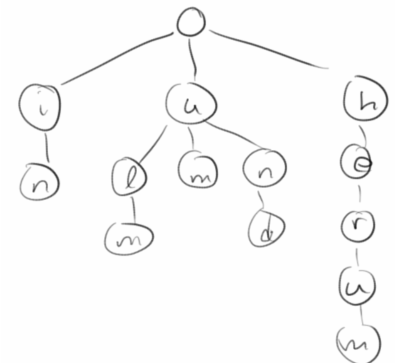
Beginnen wir mit dem Standard Trie: es handelt sich um einen geordneten Baum bei dem der Wurzelknoten leer ist. Die Knoten eines Tries enthalten jeweils einen Character. Beim Standard Trie sind die Leaves leer, das muss aber nicht sein. Wenn wir einen String einfügen, z.B. das Wort "ulm", dann fügen wir einen Buchstaben nach dem anderen in den Trie ein, von der Wurzel beginnend. Geht man den Baum von der Wurzel zu den Blättern durch, dann erhält man wieder den String den man eingefügt hat. Das interessante sind jetzt Worte bei denen die ersten Buchstaben übereinstimmen, wie z.B. "um" und "ulm" beginnen beide mit "u". Für dieses erste "u" verwendet der Trie nur einen Knoten, ist also theoretisch sehr sparsam.

Die Datenstruktur Trie unterstützt die folgenden Methoden:

- **add("stun"):** fügt einen neuen String ein;
- **startsWith("st"):** testet ob es einen String gibt der mit "st" beginnt;
- **contains("stun"):** testet ob es den String "stun" gibt;
- **nodesWithPrefix("st"):** gibt alle Strings die mit "st" beginnen;
- **prefixesThatMatch("s..n"):** gibt nur die Prefixes der Strings die dem Pattern "s..n" entsprechen;
- **nodesThatMatch("s..n"):** gibt alle Strings die dem Pattern "s..n" entsprechen.

Vor allem die letzten beiden sind wunderbar für das Suchen geeignet. Die Laufzeiteigenschaften für Suchen, Einfügen und Löschen sind proportional zur Größe des Alphabets, d, und der Länge, m, des Strings der gesucht oder eingefügt wird, also O(d*m).

Extension



Es gibt eine Erweiterung zu den Standard Tries, die es auch erlaubt sich die Position eines Strings in einem gegebenen Dokument zu merken. Dazu betrachten wir das folgende Beispiel, eine alte Binsenweisheit unter Börsenprofis:

```
in ulm und um ulm und um ulm herum
0123456789012345678901234567890123
```

Wir fügen wie gewohnt unsere Strings in den Trie ein. Zusätzlich aber verwenden wir noch die ungenutzten Leaf Knoten, um uns die Position zu merken an der der String im Text vorkommt. Damit können wir dann nicht nur sagen ob ein bestimmtes Wort in einem Text vorkommt, sondern wir können auch noch sagen wo es vorkommt. Wir können daraus sogar das ursprüngliche Dokument wieder herstellen.

Other Tries

Wenn es einige sehr lange Worte gibt, dann geht die normale Trie Datenstruktur nicht sehr sorgfältig mit dem Speicher um. Für solche Szenarien gibt es die komprimierten Tries. Bei diesen werden die Enden komprimiert. Es gibt auch noch sogenannte Suffix Tries, die bei Suchen nach Teilstrings sehr hilfreich sein können.

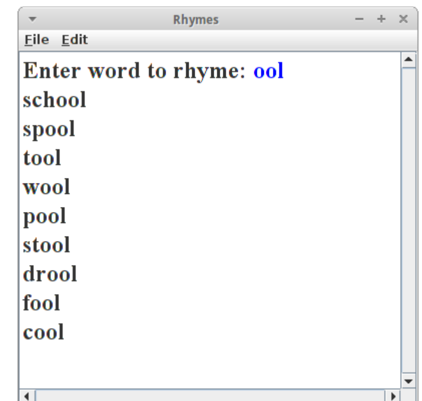
.

Review

Nach einer kurzen Einführung in die Textsuche, haben wir uns kurz Editierdistanzen und phonetische Suche angesehen. Die Datenstruktur Trie wird uns gleich noch ein wenig weiter beschäftigen.

Projekte

Die Projekte In diesem Kapitel haben's in sich. Nicht vergessen, wir sind erst im zweiten Semester!



Rhymes

Ein wirklich einfach Anwendung der *Trie* Datenstruktur ist ein Programm für angehende Dichter. Wenn wir also Worte suchen die sich auf "cool" reimen, dann suchen wir nach allen Worten die auf "ool" enden. Der Trick ist die Worte falsch herum in einen Trie zu speichern.

Wir instanziieren den *Trie*:

```
private SimpleTrie trie = new SimpleTrie();
```

und in den Trie schreiben wir einfach unser gesamtes Wörterbuch

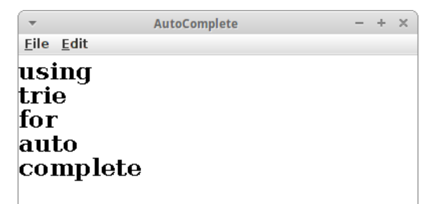
```
loadLexiconFromFile("dictionary_en_de.txt");
```

Beim Einfügen in den Trie

```
trie.add( reverseString(en.toLowerCase()) );
```

achten wir aber darauf, dass wir alle Worter falsch herum einfügen. Wenn wir dann nach Reimen suchen, ist das ganz einfach:

```
public void run() {
    loadLexiconFromFile("dictionary_en_de.txt");

    String word = readLine("Enter word to rhyme: ");
    for (String s : trie.nodesWithPrefix( reverseString(word) )) {
        println( reverseString(s) );
    }

}
```



AutoComplete

Eine Trie Datenstruktur kann man auch dazu verwenden ein Auto-Complete zu implementieren. Wir laden wieder das englische Wörterbuch wie bei den Reimen in einen Trie. Nur dieses mal drehen wir die Wörter nicht herum. Wenn der Nutzer jetzt anfängt zu tippen, dann warten wir die ersten drei Buchstaben ab, ansonsten gäbe es zu viele Möglichkeiten. Dann aber suchen wir mit

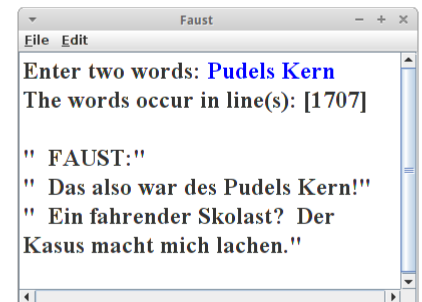
```
trie.nodesWithPrefix(text)
```

nach einem Wort, das mit diesen drei Buchstaben beginnt. Da es wahrscheinlich mehr als nur ein Wort gibt, das mit diesen drei Buchstaben beginnt, wählen wir das kürzeste aus:

```
String suggestion = "hi there i am a very long string";
for (String s : trie.nodesWithPrefix(text)) {
    if (s.length() < suggestion.length()) {
        suggestion = s;
    }

}
```

Hier könnte man bestimmt auch andere Heuristiken verwenden, aber einfach ist gut. Und das war es schon. Man könnte das Programm jetzt noch so erweitern, dass es beim Eingeben des Tab-Zeichens den momentanen Vorschlag übernimmt, und beim Eingeben des Leerzeichens den String nimmt den der Nutzer eingegeben hat, denn nicht alle Wörter sind im Wörterbuch.

Faust



Wenn man sich mit der *Trie* Datenstruktur erst einmal angefreundet hat, ist es überraschend was man mit ihr alles machen kann. Natürlich kennt jeder seinen Faust, wenigstens der Tragödie erster Teil. Aber wie war das mit dem "Pudels Kern" noch mal? Wir würden gerne wissen an welcher Stelle im Faust dieses Zitat vorkommt. Diese Art von Suche nennt man "Proximity" Suche. Wir wollen uns hier auf Suchen nach genau zwei Wörtern beschränken.

Zunächst definieren wir zwei Datenstrukturen:

```
private Trie<Integer> trie = new Trie<Integer>();

private List<String> text = new ArrayList<String>();
```

In *trie* speichern wir die Zeilennummern, und in *text* speichern wir den gesamten Text, dabei soll der Index einfach die Zeilennummer sein.

Wir lesen dann den Faust Zeile für Zeile, speichern die in die Liste *text*, und zusätzlich fügen wir sie in den *trie*: als Schlüssel verwenden wir immer zwei Wörter, *previousToken* und *token*, und als Wert die Zeilennummer:

```
private void loadLexiconFromFile(String fileName) {
    try {
        BufferedReader br = new BufferedReader(new FileReader(fileName));
        int lineNr = 1;
        while (true) {
            String words = br.readLine();
            if (words == null)
                break;
            text.add(words);
            StringTokenizer st =
                new StringTokenizer(words, " ,.:;'!?-()\"");
            String previousToken = "";
            while (st.hasMoreTokens()) {
                String token = st.nextToken().toLowerCase();
                token = token.replace("ä", "ae");
                token = token.replace("ö", "oe");
                token = token.replace("ü", "ue");
                token = token.replace("ß", "ss");
                if (previousToken.length() > 0) {
                    trie.add(previousToken + "" + token, "" + lineNr);
                }
```

```
                    previousToken = token;
                }
                lineNr++;
            }

            br.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Die Suche nach Zitaten ist dann trivial:

```
String searchWords = readLine("Enter two words: ");
String[] words = searchWords.toLowerCase().split(" ");

Set<Integer> lineNrs = trie.get(words[0].trim() + words[1].trim());

println("The words occur in line(s): " + lineNrs);
```

Wenn man dann noch die Zeilen drum herum zitieren möchte, muss man nur in der *text* Liste nachsehen:

```
for (int nr : lineNrs) {
    println("\"" + text.get(nr - 2) + "\"");
    println("\"" + text.get(nr - 1) + "\"");
    println("\"" + text.get(nr - 0) + "\"");
}
```

Interessant wäre jetzt noch zu wissen wie effektiv die Trie Datenstruktur ist, und was ihre Grenzen sind. Wenn man die kennt, wäre der nächste Schritt eine kleine Suchmaschine für die eigene Festplatte zu bauen: einfach alle Dateien durchgehen, alle Wörter einfach in den Trie schreiben, und als Wert verwendet man den Dateinamen.

JVM Monitor

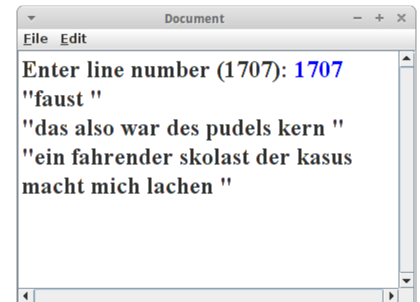
Wir wollen also wissen wieviel Speicher unsere Trie Datenstruktur verbraucht. Dazu kann man den JVM Monitor verwenden [14]. Der kommt normalerweise mit Eclipse, man kann ihn aber auch im nachhinein noch installieren. Über Window > Show View > Other findet man ihn unter "Java Monitor". Im "JVM Explorer View" sieht man alle Programme die gerade in der JVM laufen. Man startet dann das Programm das man untersuchen möchte, also z.B. unser *Faust* Programm. Dann muss man rechts-klicken und "Start Monitoring" auswählen. Das macht dann ein neues "Properties" Fenster auf (hängt manchmal ein bisschen, einfach zwischen den Fenstern hin- und herklicken). Und das sagt uns z.B., dass es von dem Trie.Node[] Array 104958 Objekte gibt, die insgesamt etwas mehr als 12 MByte verbrauchen.

Wenn wir bedenken, dass die *Faust.txt* Datei nur ca. 222 kByte groß ist, dann ist das ziemlich verschwenderisch. Allerdings scheint es linear zu skalieren und wir haben noch keinerlei Optimierungen vorgenommen.

```
File          File Size    Objects    RAM
Faust.txt      222 kByte   104958     12 MByte
Ulysses.txt   1600 kByte   617142     74 MByte
```

.

Document



In diesem Beispiel wollen wir zeigen, dass man ein ganzes Dokument in einem Trie speichern kann, und das man es auch wieder rekonstruieren kann. Wir nehmen wieder unseren Faust. Unser Trie ist der gleiche wie gehabt, lediglich beim Parsen speichern wir jetzt zusätzlich zur Zeilennummer auch noch die Position,

```
int position = 1;
StringTokenizer st = new StringTokenizer(words,
" ,.:;'!?-()\"");
while (st.hasMoreTokens()) {
    String token = st.nextToken().toLowerCase();
    // we assume lines are less than 100 chars!
    trie.add(token, lineNr * 100 + position);
    position += token.length() + 1;
}

lineNr++;
```

dabei gehen wir davon aus, dass es keine Zeilen gibt die mehr als 100 Zeichen haben. Damit ist das Laden des Tries erledigt.

Die Rekonstruktion ist allerdings nicht ganz trivial. Dabei verwenden wir folgende Datenstruktur:

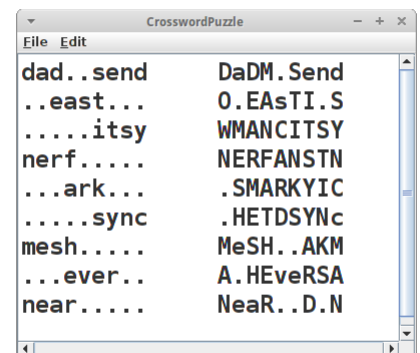
```
Map<Integer, Map<Integer, String>> document;
```

Eine Map mit einer Map. Der Key in der ersten Map ist die Zeilennummer. Der Key in der zweite Map ist die Position in der Zeile, und der String ist einfach das Wort in der Zeile und an der Position. Will man dann die Zeile Nummer 1707, sagt man einfach:

```
Map<Integer, String>> sentence = document.get(1707);
```

Und wenn es sich hier um eine TreeMap handelt kann man einfach ein Wort nach dem anderen ausgeben.

CrosswordPuzzle



Meine Mutter ist kreuzworträtselsüchtig. Und so mit der Zeit geht das ganz schön ins Geld. Also dachte ich mir, muss man doch automatisch machen können. Stellt sich heraus automatisch Kreuzworträtsel zu generieren ist überraschend kompliziert. Aber mit dem *Trie* ist es gar nicht so schwer. Der Trick hier ist die *nodesThatMatch()* Methode des Tries. Z.B., liefert der Aufruf:

```
nodesThatMatch("s..n")
```

die Resultate "stun" und "stinks", wohingegen der Aufruf

```
nodesThatMatch("s..n..")
```

nur "stinks" liefert.

Um anzufangen benötigen wir ein paar Worte:

```
private final String[] words = { "ark", "card", "dad", "day", "dear",
    "down", "east", "erase", "ever", "father", "itsy", "man", "mesh",

    "near", "nerf", "send", "stinks", "stun", "sync", "yard" }
```

Bei Kreuzworträtseln ist es sinnvoll die Worte nicht einfach aus einem Wörterbuch zu nehmen, sondern solche die thematisch und vom Schwierigkeitsgrad irgendwie passen. Uns interessieren nur die richtigen Antworten, aber natürlich bräuchte man auch irgendwo die Fragen die dazu gehören.

Die Idee ist, dass wir im ersten Schritt unser Kreuzworträtsel definieren als 9x9 Array von chars:

```
private char[][] puzzle = new char[PUZZLE_SIZE][PUZZLE_SIZE];
```

dann initialisieren wir es mit lauter Punkten '.' und verteilen zufällig ein paar unserer Wort horizontal:

```
dad..send
..east...
.....itsy
nerf.....
...ark...
.....sync
mesh.....
...ever..

near.....
```

Die Worte die wir horizontal verteilen schreiben wir in Kleinbuchstaben. Interessant wird ein Kreuzworträtsel nicht dadurch, dass die Wörter einfach nur horizontal verteilt sind, sondern auch vertikal. Wenn wir uns das Beispiel oben ansehen, bemerken wir z.B. dass da Platz wäre für ein "fa.her" (rot markiert). Die Methode *nodesThatMatch()* ist genau was wir hier brauchen, wenn wir ihr

```
nodesThatMatch("fa.her")
```

übergeben, würde die uns "father" zurückliefern. Das war's eigentlich schon. Wir probieren einfach ein paar vertikale Wörter, zufällig, und schauen was rauskommt:

```
DaDM.Send
AEEAsTI.S
YVANCITSY
nERFANSTN
.RMARKYIC
..ETDSYNC
MeSH..AKA
A.HEveRSR

NeaR..D.D
```

Passt doch. Unser Algorithmus ist nicht gerade besonders optimiert, er wird daher auch nur für relative kurze Worte funktionieren. Eine interessante Übung wäre es das Laufzeitverhalten dieses Algorithmus abzuschätzen.

Longest Common Substring

Sehr häufig müssen wir testen ob sich zwei Strings "ähnlich" sind. Zum Beispiel beim Programmieren, möchte man manchmal den Unterschied zwischen einer älteren und neueren Versionen wissen (z.B. UNIX diff command), oder wenn mehrere Personen an einem längeren Text arbeiten, möchte man wissen was sind denn die Änderungen zwischen verschiedenen Versionen des Dokuments. In der Wikipedia gibt es ab und an mal sogenannte "Editing Wars". Auch da möchte man gerne wissen was hat sich denn zur Vorgängerversion geändert. Und schließlich eine ganz wichtige Anwendung ist in der DNA Analyse: hier interessieren einen sehr häufig die Unterschiede zwischen zwei verschiedenen DNA Sequenzen.

Wenn wir einen Brute-Force Algorithmus verwenden um zwei Strings und all ihre Substrings zu vergleichen, dann funktioniert das nur für relativ kurze Strings, denn das Laufzeitverhalten dieses Algorithmus ist exponentiell, als $O(2^n)$. Zu unserem Glück gibt es aber einen Algorithmus der das Problem in polynomischer Zeit schaffen kann, und zwar O(m*n), wobei *m* die Länge des einen und *n* die Länge des anderen Strings ist. Der Algorithmus ist eigentlich ganz einfach, und wir wollen ihn an einem genetischen Beispiel demonstrieren.

Homer möchte wissen, ob Bart wirklich sein Sohn ist. Also hat er ein Haar von sich und eines von Bart geschnappt und an den DNA Shop geschickt. Zurück kam folgendes:

```
Homer = "GTTCCTAATA"
Bart = "CGATAATTGAGA".
```

Was Homer jetzt machen muss, ist ein Stück kariertes Papier hernehmen, seinen DNA String entlang der x-Achse ausschreiben, und den von Bart entlang der y-Achse. Danach geht er einfach Zeile für Zeile durch und macht einfach in jedes Kästchen ein Kreuzchen wo beide Strings den gleichen Buchstaben haben. Wenn er dann die Kästchen die sich zu Diagonalen zusammenschließen, markiert, dann findet er die Übereinstimmungen in den beiden DNAs. Man sucht also nach dem *größten gemeinsamen Teilstring*. Ich würde mal sagen, dass grob 50% der DNA übereinstimmen, also alles in Ordnung.

Longest Common Subsequence

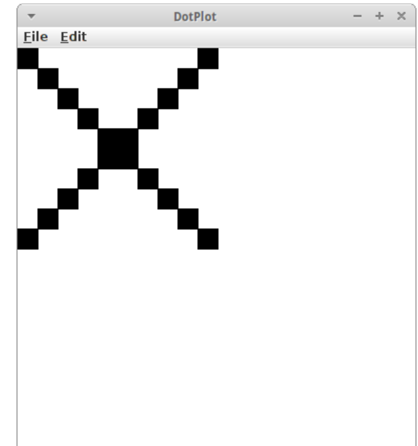
Während wir oben nach einem exakten Match gesucht haben, erlauben wir bei der Suche nach der *Longest Common Subsequence* [18] auch Fehler:

Der Algorithmus ist sehr ähnlich, mit dem gleichen Laufzeitverhalten. Man trägt wieder die beiden zu vergleichenden Strings an der x- und y-Achse auf. Im Unterschied zum obigen Beispiel benutzt man jetzt aber einen Zähler, den man am Anfang auf 0 setzt.

1. man fängt oben links an, und schreibt in die nullte Zeile und Spalte lauter Nullen;
2. man geht dann entlang der x-Achse, und jedes Mal wenn die Buchstaben übereinstimmen, erhöht man den Zähler um eins;
3. in der nächsten Zeile geht wieder entlang der x-Achse. Jetzt kommt es darauf an ob die Buchstaben übereinstimmen, dann wird wieder hochgezählt, oder ob der Wert in der Zelle davor oder darüber höher ist. Es wird immer der höhere Wert übernommen;
4. das Ganze macht man so lange bis man alle Zeilen durch hat.

Den Wert den der Zähler am Ende hat, ist die *längste gemeinsame Teilsequenz*. Die Sequenz selbst erhält man, wenn man der Diagonale folgt, in der die Werte des Zählers höher werden. Ein sehr schöne Animation der ganzen Prozedur findet man auf Youtube [19].

DotPlot



Der Dotplot (dt. Punktauftragung) [20] ist eine Anwendung des *Longest Common Substring* Algorithmus, der vor allem in der Bioinformatik sehr populär ist [21]. Die visuelle Darstellung ist wirklich super-easy, und das kann sogar ein Erstsemester. Wir bitten unseren User einfach zwei Strings einzugeben:

```
public void run() {
    IODialog dialog = new IODialog();
    String s1 = dialog.readLine("Enter first
string:");
    String s2 = dialog.readLine("Enter second
string:");
    showSimilarity( s1, s2 );

}
```

die wir dann in der Methode *showSimilarity()* visuell darstellen:

```
private void showSimilarity(String s1, String s2) {
    for (int i = 0; i < s1.length(); i++) {
        for (int j = 0; j < s2.length(); j++) {
            if ( s1.charAt(i) == s2.charAt(j) ) {
                GRect pixel = new GRect(BLOCK_SIZE, BLOCK_SIZE);
                pixel.setFilled(true);
                add( pixel, i*BLOCK_SIZE, j*BLOCK_SIZE );
            }
        }
    }

}
```

Wir haben zwei verschachtelte Schleifen, die eine geht durch alle Buchstaben des ersten Strings, und die zweite durch alle Buchstaben des zweiten Strings, und falls die beiden Buchstaben gleich sind, zeichnen wir einfach ein kleines Rechteck.

Wenn wir für den ersten String und den zweiten String das gleiche Wort eingeben, dann kommt einfach eine gerade Linie heraus. Interessant wird's wenn wir mal ein Palindrom eingeben. Dann kommt da ein 'X' heraus. Auch Fast-Palindrome geben interessante Muster:

- abcdefgh
- lagerregal
- abracadabra
- max i stay away at six am

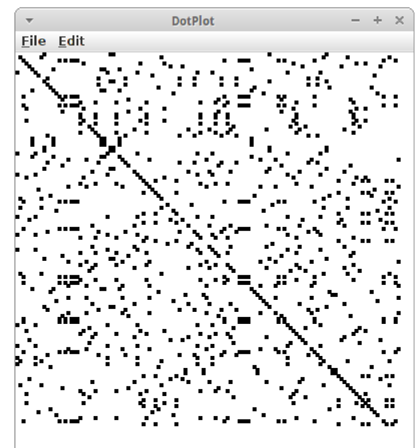
Es ist ziemlich offensichtlich, dass diese visuelle Darstellung hilft bestimmte Regelmäßigkeiten auf einfach Art und Weise zu entdecken.

Gehen wir aber einen Schritt weiter, denn wir wollen ja den *Longest Common Substring* finden. Glücklicherweise finden wir in der Wikipedia Pseudocode der sich ganz einfach in Java übertragen lässt [17]:

```
private String findLargestCommonSubstring(String S, String T) {
    int[][] L = new int[S.length()][T.length()];
    int z = 0;
    int endIndex = 0;
    for (int i = 0; i < S.length(); i++) {
        for (int j = 0; j < T.length(); j++) {
            if (S.charAt(i) == T.charAt(j)) {
                if (i == 0 || j == 0) {
                    L[i][j] = 1;
                } else {
```

```
                    L[i][j] = L[i-1][j-1] + 1;
                }
                if ( L[i][j] > z ) {
                    z = L[i][j];
                    endIndex = i;
                }
            } else {
                L[i][j] = 0;
            }
        }
    }
    if ( z > 0 ) {
        return S.substring( endIndex-z+1, endIndex+1 );
    }
    return null;

}
```

Die Länge der Übereinstimmung des Strings ist ein Meßgröße dafür wie ähnlich sich die beiden Strings sind.



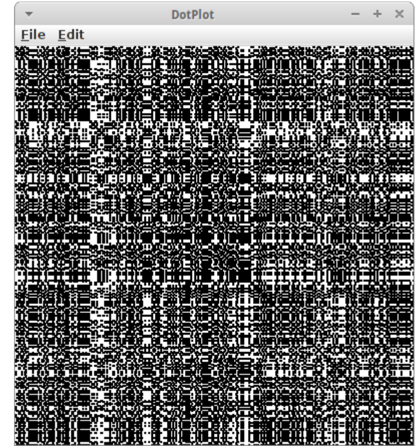
Horse, Minke Whale and Kangaroo

Als angehende Bioinformatiker würden wir gerne die phylogenetische Beziehungen zwischen Pferd, Zwergwal und rotem Känguru bestimmen. Im internet findet man "The Basic Local Alignment Search Tool (BLAST)" [22] und wenn man dort "horse ribonuclease pancreatic" als Suchbegriff eingibt findet man:

```
horse
horse ribonuclease pancreatic
>sp|P00674|1-128
KESPAMKFERQHMDSGSTSSSNPTYCNQMMKRRNMTQGWCKPVNTFVHEPLADVQAICLQ
KNITCKNGQSNCYQSSSSMHITDCRLTSGSKYPNCAYQTSQKERHIIVACEGNPYVPVHF
DASVEVST

minke whale
balac ribonuclease pancreatic
>sp|P00673|1-124
RESPAMKFQRQHMDSGNSPGNNPNYCNQMMMRRKMTQGRCKPVNTFVHESLEDVKAVCSQ
KNVLCKNGRTNCYESNSTMHITDCRQTGSSKYPNCAYKTSQKEKHIIVACEGNPYVPVHF
DNSV

red kangaroo
macru ribonuclease pancreatic
>sp|P00686|1-122
ETPAEKFQRQHMDTEHSTASSSNYCNLMMKARDMTSGRCKPLNTFIHEPKSVVDAVCHQE
NVTCKNGRTNCYKSNSRLSITNCRQTGASKYPNCQYETSNLNKQIIVACEGQYVPVHFDA
YV
```

Wenn man diese Daten einfach mittels unseres DotPlot Programms anzeigen lässt, erkennt sogar der Laie, dass es da eine Beziehung zwischen den dreien gibt.

Jurassic Park



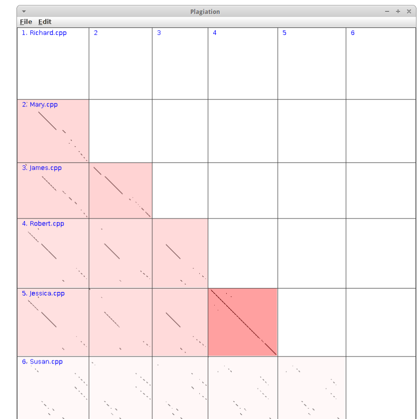
Kommen wir zu unserem letzten Beispiel aus der Genanalyse: wir würden gern wissen wie nahe sind die heute lebenden Elephanten, also der afrikanische und der indische, mit dem Mammut verwandt [23]? Ein paar der DNA Schnipsel findet man auf den Webseiten des National Center for Biotechnology Information [24]. Dort kann man in der Nucleotide Datenbank suchen. Unter den folgenden Kürzeln [23],

```
African elephant: DQ316069
Asiatic elephant: DQ316068
Woolly mammoth: DQ316067
N. American mastodon: EF632344
Rock hyrax: NC_004919

Dugong: NC_003314
```

findet man DNA Schnipsel der entsprechenden Spezies. Wenn wir die einfach mit unserem simplen DotPlot Programm laden, dann sieht man zwar eine Diagonale, aber man sieht auch viel Unsinn. Das hat damit zu tun, dass mit den Buchstaben "ACGT" eben keine besonders große Vielfalt im DNA Alphabet exisitiert. Was die Genetiker hier machen, sie zeichnen nur dann Punkte, wenn die diagonalen Linien eine gewisse Mindestlänge haben, z.B., nur wenn mindestens sechs Aminosäuren übereinstimmen zeichnen wir diese. Wenn wir wollen könnten wir unser DotPlot Programm so verbessern, dass es genau das tut.

Challenges

Plagiation



Eine weitere interessante Anwendung für den *Longest Common Substring* Algorithmus, ist die Erkennung von Plagiaten. Wir wollen die Hausarbeiten von sechs Studierenden (Richard, Mary, James, Robert, Jessica und Susan) vergleichen. Dazu generieren wir die DotPlots für alle möglichen Kombinationen, also 2 aus 6. Diese kombinieren wir dann zu einer gemeinsamen großen Übersichtsgrafik: dort tragen wir die Namen der Studierenden jeweils an der x- und y-Achse auf, und zeichnen in der jeweiligen Zelle den DotPlot zwischen den Hausarbeiten der beiden Studierenden.

In der Übersichtsgrafik kann man sofort erkennen, dass Jessica und Robert zusammen gearbeitet haben, während Susan alleine gearbeitet hat. Das sieht man an den langen, durchgezogenen Diagonalen. Je länger oder je mehr, desto größer ist die Übereinstimmung. Bei den anderen sieht man unterschiedlich große Übereinstimmungen, die natürlich teilweise auch zufällig sein können.

Um die Übereinstimmungen visuell noch stärker hervortreten zu lassen, kann man die jeweiligen DotPlots noch mit einer Hintergrundfarbe versehen. Weiß bedeutet so viel wie keine Übereinstimmungen, während rot hingegen eine große Übereinstimmung andeutet. Wie misst man denn die Übereinstimmung? Hier gibt es mehrere Ansätze: der einfachste, man nimmt einfach die Länge des längsten gemeinsamen Substrings, und teilt ihn durch die Länge des Gesamtstrings. Man könnte auch mehrere kleine Übereinstimmungen nehmen, deren Länge aufaddieren und dann skalieren. Interessanterweise funktioniert die einfach Variante aber bereits recht gut.

Eine interessante Frage die man sich stellen sollte ist, wie ist denn das Laufzeitverhalten dieses Projektes?

Research

Zu diesem Kapitel gibt es sehr interessante Research, wenigstens wenn man sich dafür interessiert wie eine Suchmaschine funktioniert.

diff

Der diff Utility vergleicht zwei Dateien und zeigt die Unterschiede an. Es basiert auf der Lösung der längsten gemeinsamen Teilsequenz. Weitere Details finden sich in Referenz [25].

Google

Wie funktioniert denn eine Suchmaschine? Gerade im Faust Beispiel haben wir eigentlich schon gesehen wie. Man scannt eben nicht nur ein Dokument, sondern sehr viele, z.B. das Web. Die Dokumente sind in dem Fall HTML Seiten. Die bereitet man etwas auf, also man entfernt alles was unnötig ist, wie HTML Tags, Stopwords und man vereinfacht die Grammatik. Was dann übrigbleibt speichert man in einem Trie. Allerdings speichert man nicht die Zeilenzahl, sondern die Webseite als Wert. Falls es mehrere Webseiten zu einem Wert gibt, dann speichert man eine Liste (oder Set) von Webseiten. Diesen Trie nennt man auch manchmal "Inverted Index" [15]. Je nachdem wieviele Seiten man hat, passt dieser Trie nicht mehr in den RAM, da muss man sich dann Gedanken machen wie man das am besten macht mit Hilfe von Festplatten. Was noch fehlt ist natürlich ein Ranking, dazu dürfte die Lektüre von Referenz [16] ganz interessant sein. Der Rest ist Geschichte, wie es so schön heißt.

Suffix Tries

Für die Suche sind *Suffix Tries* sehr wichtig. Man sollte mal im Internet nach ihnen suchen.

Fragen

1. In welchem Kontext wird der Boyer-Moore-Algorithmus verwendet?

2. Hat ein Trie genügend informationen, um den Ursprungstext komplett zu rekonstruieren? Ist es effizienter, den Text in Form eines Trie oder als Klartext zu speichern?

3. In einem der Aufgaben schrieben wir ein kleines Programm, um unseren Dichterfreunden zu helfen, bessere Reime zu schaffen. Welche Datenstruktur haben wir benutzt, um gute Reime zu finden?

4. In den Übungen sahen wir ein Beispiel, wie man Plagiate identifiziert, oder genauer, wie man Copy-Paste Plagiate entdeckt. Welchen Algorithmus würden Sie verwenden, um Plagiate zu erkennen?

5. Für die folgenden Strings konstruieren Sie einen Standard-Trie:
   - idea, idiot, idle, hit, hour, house
   - humid, stupid, stop, hungry, bear, hunger, stock
   - Fischers Fritz fischt frische Fische, frische Fische fischt Fischers Fritz.

6. Bei der Rechtschreibprüfung (spell checker) geht es darum festzustellen ob ein Wort richtig geschrieben wurde. Welche Datenstruktur ist dafür am besten wenn es sich um eine etwas kompliziertere Sprache, wie z.B. das Deutsche, handelt?

7. Vergleichen Sie die beiden Strings "GCTTCGTAACT" und "CTATGATACTG". Finden Sie den größten gemeinsamen Teilstring.

8. Was ist der Unterschied zwischen "longest common substring" (längste gemeinsamer Teilstring) und "longest common subsequence" (längste gemeinsame Teilsequenz)?

Referenzen

In diesem Kapitel gibt es ungewöhnlich viele Referenzen.

[1] The Boyer-Moore Fast String Searching Algorithm, www.cs.utexas.edu/users/moore/best-ideas/string-searching/

[2] Rabin-Karp Algorithm, Searching for Patterns | Set 3 (Rabin-Karp Algorithm), www.geeksforgeeks.org/searching-for-patterns-set-3-rabin-karp-algorithm/

[3] Knuth Morris Pratt (KMP), Searching for Patterns | Set 2 (KMP Algorithm), www.geeksforgeeks.org/searching-for-patterns-set-2-kmp-algorithm/

[4] String metric, https://en.wikipedia.org/wiki/String_metric

[5] Levenshtein distance, https://en.wikipedia.org/wiki/Levenshtein_distance

[6] Hamming distance, https://en.wikipedia.org/wiki/Hamming_distance

[7] amatch - Approximate Matching Extension for Ruby, https://github.com/makaroni4/amatch

[8] Soundex, https://en.wikipedia.org/wiki/Soundex

[9] Apache Commons Codec 1.10 API, https://commons.apache.org/proper/commons-codec/apidocs/overview-summary.html

[10] Kölner Phonetik, https://de.wikipedia.org/wiki/Kölner_Phonetik

[11] Using Fuzzy Matching to Search by Sound with Python, Doug Hellmann, informit.com/articles/article.aspx?p=1848528

[12] Data Structures and Algorithms in Java, M.T. Goodrich and R. Tamassia

[13] Tries, algs4.cs.princeton.edu/52trie/

[14] JVM Monitor, jvmmonitor.org/index.html

[15] Inverted index, https://en.wikipedia.org/wiki/Inverted_index

[16] The Anatomy of a Large-Scale Hypertextual Web Search, Engine Sergey Brin and Lawrence Page, infolab.stanford.edu/~backrub/google.html

[17] Longest common substring problem, https://en.wikipedia.org/wiki/Longest_common_substring_problem

[18] Longest common subsequence problem, https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

[19] Longest common subsequence algorithm -- example, https://www.youtube.com/watch?v=P-mMvhfJhu8

[20] Dot plot (bioinformatics), https://en.wikipedia.org/wiki/Dot_plot_(bioinformatics)

[21] Introduction to Bioinformatics, A.M. Lesk, Oxford University Press, 2005

[22] The Basic Local Alignment Search Tool (BLAST), www.uniprot.org/blast/

[23] The evolution of mammoths and their living relatives - DNA to Darwin, www.dnadarwin.org/casestudies/10/FILES/MammothSG2.0.pdf

[24] National Center for Biotechnology Information, https://www.ncbi.nlm.nih.gov/nuccore/

[25] diff utility, https://en.wikipedia.org/wiki/Diff_utility

Text

.

Techniques

Wir haben bisher schon einige algorithmische Techniken gesehen und uns ausführlich mit ihnen beschäftigt, als da waren die Iteration, die Rekursion, der Divide and Conquer Algorithmus und die Dynamische Programmierung. Diese Techniken sind sozusagen unsere Basics, die brauchen wir immer. Darüber hinaus gibt es aber noch andere Klassen von Algorithmen, die sich als sehr hilfreich herausgestellt haben. Meist arbeiten diese mit den anderen zusammen, sind also nicht unabhängig von ihnen.

Eine Methode die im Prinzip immer funktioniert ist *Brute Force*, d.h., einfach alle Möglichkeiten ausprobieren. Das Problem mit Brute Force ist wenn es viele Möglichkeiten gibt. Dann kann das Ausprobieren nämlich sehr lange dauern. Es gibt aber glücklicherweise auch noch andere Verfahren, als da sind *Greedy* Algorithmen, *Back Tracking* Algorithmen und *Randomized* Algorithmen. Natürlich gibt es auch noch andere, wir beschränken uns aber auf diese. Diese Algorithmen funktionieren nicht immer, und sie finden nicht notwendigerweise die beste Lösung, aber sehr häufig finden sie akzeptable Lösungen, und das genügt meistens.

Greedy Algorithms

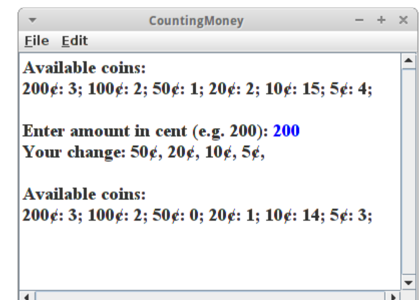
Greedy Algorithmen, auch gierige Algorithmen genannt, sind Algorithmen die bei jedem möglichen Schritt immer die gerade am besten erscheinende Alternative wählen. Ein ganz einfaches Beispiel ist der Selection Sort: der sagt einfach "suche nach der kleinsten Zahl und setze sie an den Anfang". Die Greediness besteht hier darin die kleinste Zahl zu nehmen. Greediness bedeutet es muss irgendein Auswahl- oder Bewertungkriterium geben. Zwei andere Beispiele für Greedy Algorithmen die wir schon gesehen haben sind der Dijkstra, der Prim und der Kruskal Algorithmus. Das sind Beispiele wo der Greedy Algorithmus immer eine Lösung liefert. Das muss aber nicht immer so sein.

Andere Beispiele die sich mit einem Greedy Algorithmus lösen lassen sind:

- Rucksackproblem [3]
- Problem des Handlungsreisenden
- Huffman coding [1]
- Greedy coloring [2]
- Tic-Tac-Toe [3]
- Graph Layout
- Optimales Kommunikationsnetz [3]

Wir wollen uns die Technik etwas genauer anhand von ein paar Beispielen ansehen.

CountingMoney



Für einen automatischen Kaffeeautomaten wollen wir einen Algorithmus entwickeln, der das richtige Rückgeld berechnet. Unser Automat ist gefüllt mit einer bestimmten Anzahl von verschiedenen Euro und Cent Münzen. Wir nehmen einfach mal an, unser Automat hätte vier 5-Cent-Münzen, fünf 10-Cent-Münzen, zwei 20-Cent-Münzen, eine 50-Cent-Münze, zwei 1-Euro-Münzen und drei 2-Euro-Münzen. Ein Kaffee soll 1,15 Euro kosten. Wenn jetzt ein Kaffeesüchtiger eine 2-Euro-Münze einwirft, dann soll unser Algorithmus das richtige Wechselgeld liefern, also je eine 50-Cent, 20-Cent, 10-Cent- und 5-Cent Münze.

Brute-Force

Ein möglicher Algorithmus wäre ein *Brute Force* Algorithmus [14]: der probiert einfach alle möglichen Münzkombinationen aus, bis 85 Cent rauskommt. Wieiviele Möglichkeiten gibt es? Wenn wir 17 Münzen haben (wie im Beispiel oben), also

```
Coins =
 { 5, 5, 5, 5, 10, 10, 10, 10, 10, 20, 20, 50, 100, 100, 200, 200, 200 }
```

dann gibt es 2^n, also 2^{17} oder 131072 mögliche Kombinationen. Wenn unser Computer für die Berechnung einer Kombination eine Millisekunde benötigen würde, dann müssten wir ungefähr zwei Minuten warten bis wir unser Wechselgeld bekommen würden. Hätten wir aber z.B. 30 Münzen in unserem Automaten, dann müssten wir 12 Tage auf unser Wechselgeld warten, der Kaffee wäre dann schon kalt! Also das taugt nicht.

Greedy

Die greedy Version des Algorithmus, macht eigentlich genau das, was wir normalerweise auch machen würden, wenn wir Geld zurückgeben sollen: wir suchen nach der größten Münze. Also wenn wir 85 Cent zurückgeben sollen, dann würden mit der 50 Cent Münze anfangen, und nicht mit den 5 Cent Münzen. Danach würden wir die 20 Cent, dann die 10 und schließlich die 5 Cent Münze nehmen, also

```
50¢, 20¢, 10¢, 5¢
```

Der Greedy Algorithmus für die Münzrückgabe funktioniert demnach wie folgt:

1. nimm die größte Münze die Du hast;
2. subtrahiere sie vom Rückgeld;
3. solange der Betrag größer gleich null ist
    1. gib die Münze aus und
    2. ziehe den Wert vom Rückgeld ab und gehe zurück zu 2.
4. wenn der Betrag kleiner als null ist, dann gehe zur nächst kleineren Münze und gehe zurück zu 2.

Bevor wir das in Code umsetzen, ein paar Fragen: Funktioniert der Algorithmus immer? Gibt der Algorithmus immer die geringste Anzahl von Münzen zurück? Was passiert wenn nicht genug Münzen im Automaten sind?

Wir werden zwei Versionen vorstellen wie man den Algorithmus umsetzen kann. Die erste verwendet Arrays, was wir ja eigentlich nicht tun sollen, aber didaktisch ist es damit viel einfacher zu verstehen. Wir haben ein Array mit den Münzwerten, also 200 steht für den Wert einer zwei Euro Münze, usw.:

```
private int[] coinValues = { 200, 100, 50, 20, 10, 5 };
```

und dann haben wir noch ein Array für die Anzahl der jeweiligen Münzen,

```
private int[] coinNumbers = { 3, 2, 1, 2, 15, 4 };
```

also wir haben 3 von den zwei Euro Münzen, usw. Ganz wichtig ist hier die Reihenfolgen, also einmal dass das erste Element in beiden Arrays sich auf die zwei Euro Münze bezieht. Und die Reihenfolge ist auch wichtig für die Greediness: wir wollen ja bei der Rückgabe immer mit der größten Münze beginnen, deswegen sind die Münzen sortiert in absteigender Reihenfolge. Wenn wir die Reihenfolge ändern würden, wäre der Algorithmus nicht mehr greedy.

Mit diesen Voraussetzungen ist der eigentliche Algorithmus recht überschaubar:

```
private void giveChange(int amount) {
    for (int i = 0; i < coinValues.length; i++) {
        while ((coinNumbers[i] > 0) && ((amount - coinValues[i]) >= 0)) {
            amount = amount - coinValues[i];
            coinNumbers[i]--;
            print(coinValues[i] + "¢, ");
        }
    }

}
```

Kommen wir zur zweiten Version: hier verwenden wir anstelle zweier Arrays eine *TreeMap*:

```
private TreeMap<Integer, Integer> coins;
```

Tree deswegen weil wir möchten, dass die Münzen der Größe nach sortiert sind. Der Key in der Map ist der Münzwert, und der Value ist die Anzahl der Münzen. Wir befüllen unsere Kaffeemaschine mit Geld in der *initMap()* Methode:

```
private void initMap() {
    // this is a dirty trick, to reverse the order the treemap is
traversed:
    coins = new TreeMap<Integer, Integer>(Collections.reverseOrder());

    coins.put(5, 4); // four 5-cent coins
    coins.put(10, 15); // fifteen 10-cent coins,
    coins.put(20, 2); // two 20-cent coins,
    coins.put(50, 1); // one 50-cent coin,
    coins.put(100, 2); // two 1-euro coins,
    coins.put(200, 3); // three 2-euro coins

}
```

wobei wir der TreeMap explizit sagen müssen, dass die Münzen absteigend und nicht aufsteigend sortiert werden sollen.

Auch hier ist dann der eigentliche Algorithmus sehr überschaubar:

```
private void giveChange(int amount) {
    for (int coin : coins.keySet()) {
        int nrOfCoins = coins.get(coin);
        while ((nrOfCoins > 0) && ((amount - coin) >= 0)) {
            amount = amount - coin;
            nrOfCoins--;
            print(coin + "¢, ");
        }
        coins.put(coin, nrOfCoins);
    }

}
```

Soviel zum Kaffeeautomaten.

Scheduling

Wir wollen einen Greedy Algorithmus entwickeln der unseren Terminkalender organisiert. Dabei beschränken wir uns auf eine Woche. Die Woche unterteilen wir in Zeitschlitze, z.B. jeweils einstündige. Der Tag beginnt im 6 Uhr morgens und endet um 24 Uhr.

Als Erstes generieren wir eine Liste von *Tasks*, also Aufgaben, die wir erledigen müssen:

- Frühstück, dauert 1h, hohe Priorität, jeden Tag, um 7:00 Uhr;
- Programmieren 2 Vorlesung, dauert 2h, mittlere Priorität, Fretag, um 8:00 Uhr;
- Party, dauert 4h, niedrige Priorität, beliebiger Tag, beliebige Uhrzeit.

Der zweite Schritt ist dann diese Aufgabenliste zu nehmen und auf die Woche so zu verteilen, dass es möglichst keine Konflikte gibt.

Auch hier könnte man wieder einen *Brute Force* Ansatz wählen, aber ohne ins Detail zu gehen, dürfte schnell klar sein, dass wir auch hier wieder sehr lange auf unsere Antwort warten müssten.

Für den Greedy Ansatz benötigen wir wieder irgendein Auswahl- oder Bewertungkriterium mit dem wir Prioritäten setzen können, und natürlich bieten sich die Prioritäten (also Frühstück hohe Priorität) dafür an.

Wir beginnen damit, dass wir aus unseren *Tasks* erst einmal eine Klasse machen:

```
private class Task implements Comparable {
    protected String name;  // name of task, eg "breakfast", "prog2", ...
    protected int duration; // in hours
    protected int priority; // 0 is high, 1 is medium, 2 is low
    protected String day;   // "EveryDay", "Friday", "AnyDay"
    protected int time;     // 7 for 7:00 am or 13 for 1pm or -1 for
anytime

    public Task(String name, int duration, int priority, String day, int
time) {
        super();
        ...
    }

    public int compareTo(Object o) {
        if (o instanceof Task) {
            return this.priority - ((Task) o).priority;
        }
        return 0;
    }

}
```

Die Klasse hat also die gewünschten Attribute. Aus den Prioritäten haben wir Zahlen gemacht. Es stört vielleicht, dass die Attribute *protected* sind, da die Klasse aber private ist, und es sich um eine lokale Klasse handelt ist das nicht so schlimm. Wir können mal ein paar Tasks anlegen:

```
new Task("Buy Food", 4, 2, "AnyDay", -1);
new Task("Breakfast", 1, 0, "EveryDay", 7);

new Task("Prog 2", 2, 1, "Friday", 8);
```

Kommen wir zum *Comparable* Interface: wir wollen zwei Tasks vergleichen können, und zwar über ihre Prioritäten. Das machen wir in der *compareTo()* Methode: die gibt Null (also 0) zurück wenn die beiden Tasks gleiche Priorität haben, eine positive Zahl falls unsere Priorität höher ist, und andernfalls eine negative Zahl.

Dieses Interface benötigen wir, weil wir mit einer *PriorityQueue* arbeiten wollen, und zwar einer *PriorityQueue* von Tasks:

```
private PriorityQueue<Task> tasks = new PriorityQueue<Task>();
```

Eine PriorityQueue ist wie eine Queue, also first-in-first-out, aber zusätzlich berücksichtigt sie noch die Prioritäten, d.h. kommt ein zusätzlicher Task, der aber eine höhere Priorität hat, dann kommt der an den Anfang der Queue. Ist so ein bisschen wie mit den Sanitätern. Wir befüllen dann einfach unsere Queue mit den Tasks:

```
tasks.add(new Task("Buy Food", 4, 2, "AnyDay", -1));
tasks.add(new Task("Breakfast", 1, 0, "EveryDay", 7));

tasks.add(new Task("Prog 2", 2, 1, "Friday", 8));
```

Die Greediness kommt in diesem Beispiel also zum Einen über die Priorität, aber zum Anderen auch über die Reihenfolge mit der wie die Tasks in die Queue einfügen. Das Hinzufügen geschieht in zwei Schritten. Erst einmal gehen wir alle Tasks durch:

```
Task task;
while ((task = tasks.poll()) != null) {

    if (task.day.contains("EveryDay")) {
        for (int i = 0; i < weekdays.length; i++) {
            addToCalendar(i, task);
        }

    } else if (task.day.contains("AnyDay")) {
        addToCalendar(task);

    } else {
        for (int i = 0; i < weekdays.length; i++) {
            if (task.day.contains(weekdays[i])) {
                addToCalendar(i, task);
            }
        }
    }

}
```

und versuchen für jeden einen Platz im Kalendar zu finden. Die normale addToCalendar() Methode versucht den Task an den vorgegeben Tag und Uhrzeit einzufügen,

```
private void addToCalendar(int day, Task task) {
    if (!insertTaskIntoCal(day, task.time, task)) {
        println("Conflict: " + task);
    }

}
```

während die allgemeinere *addToCalendar()* Methode, die bei AnyDay aufgerufen wird, irgendeinen Zeitschlitz versucht zu finden:

```
private void addToCalendar(Task task) {
    int day = 0;
    int hour = 7;
    while (!insertTaskIntoCal(day, hour, task)) {
        hour++;
        if (hour > 23) {
            day++;
            hour = 7;
            if (day > 4) {
                System.out.println("Conflict: " + task);
                break;
            }
        }
    }

}
```

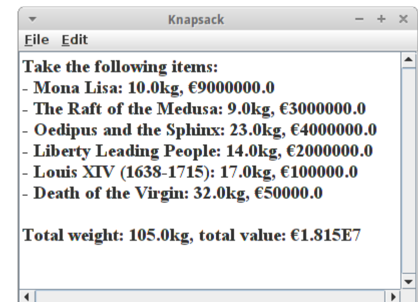
Das eigentliche Einfügen passiert in der *insertTaskIntoCal()* Methode:

```
private boolean insertTaskIntoCal(int day, int hour, Task task) {
    boolean success = true;
    // test if insert is possible
    for (int i = 0; i < task.duration; i++) {
        if (calendar[day][hour + i] != null) {
            success = false;
        }
    }
```

```
        // do the insert
        if (success) {
            for (int i = 0; i < task.duration; i++) {
                calendar[day][hour + i] = task.name;
            }
        }
        return success;

    }
```

Unser Greedy Algorithmus funktioniert überraschend gut. Allerdings kann er im Gegensatz zum Backtracking Algorithmus einmal getroffene Entscheidungen nicht mehr rückgängig machen. D.h., wenn er sich einmal verrannt hat, dann muss man von Hand nachhelfen.

Das Scheduling Problem gibt es in ganz vielen verschiedenen Varianten, und in Referenz [4] wird die Version des Interval Scheduling sehr anschaulich behandelt.



Knapsack

Beim Knapsack Problem, auch Rucksackproblem genannt, geht es darum nach einem gewissem Maximierungskriterium z.B. einen Rucksack zu befüllen. Angenommen unser Meisterdieb (Daniel "Danny" Ocean from Ocean's Eleven) will den Louvre ausrauben. Sein Team kann aber nur 300 kg tragen. Deswegen hat er uns als Computer-Experten geheuert, damit wir ihm sagen was er mitnehmen soll, um seinen Profit zu maximieren.

Danny hat im Internet eine Liste von Kunstwerken gefunden die es im Louvre gibt:

Artwork	**Artist**	**Type**	**Weight / kg**	**Price / euro**
Venus de Milo	Found in Melos	sculpture	315	20000000
Mona Lisa	Leonardo da Vinci	painting	10	9000000
Liberty Leading the People	Eugene Delacroix	painting	14	2000000
Psyche Revived By the Kiss of Love	Antonio Canova	sculpture	545	5000000
Oedipus and the Sphinx	Jean-Auguste-Dominique Ingres	painting	23	4000000
The Raft of the Medusa	Theodore Gericault	painting	9	3000000
Milon de Crotone	Pierre Puget	sculpture	332	4000000
Louis XIV (1638-1715)	Hyacinthe Rigaud	painting	17	100000
Death of the Virgin	Michelangelo Merisi	painting	32	50000
Cy Twombly's Ceiling	The Louvre's Cy Twombly Ceiling	ceiling	10000	10000

(Die Liste der Kunstwerke stammt aus [5], das Gewicht und die Preise sind rein zufällig.)

Brute-Force

Wieder kann dieses Problem durch Brute-Force gelöst werden, d.h. einfach alle möglichen Kombinationen ausprobieren. Das findet mit Sicherheit die beste Lösung. Das Problem ist, dass es 2^n mögliche Kombination gibt und damit dieser Ansatz sehr schnell sehr lange dauert.

Greedy

Die zweite Möglichkeit ist wieder mal einen Greedy Ansatz zu wählen. Wir benötigen natürlich irgendein Auswahl- oder Bewertungkriterium mit dem wir Prioritäten setzen können. Wir haben drei Möglichkeiten:

- wir könnten einfach nach dem Preis, also dem Profit maximieren;
- wir könnten nach dem Gewicht maximieren, also z.B. nur die leichten Sachen mitnehmen;
- oder wir könnten nach der *Profit-Density* maximieren, also dem Verhältnis Preis zu Gewicht.

Alle drei Ansätze können zu einer anständigen Lösung führen, aber es ist nicht garantiert, dass es wirklich die beste Lösung ist.

Wenn wir das in Code umsetzen wollen, benötigen wir erst wieder eine Klasse um die Kunstwerke zu repräsentieren:

```
private class Valuable {
    protected String name;
    protected double weight;
    protected double price;

    public Valuable(String name, double weight, double price) {
        this.name = name;
        this.weight = weight;
        this.price = price;
    }

    public String toString() {
        return name + ": " + weight + "kg, €" + price + "";
    }
}
```

Für uns relevante Daten sind der Name, das Gewicht und der geschätzte Preis des Kunstwerks. Wie bei CountingMoney2 verwenden wir wieder eine TreeMap als Datenstruktur:

```
private TreeMap<Double, Valuable> valuables;
```

die wir in der *initMap()* Methode initialisieren:

```
private void initMap() {
    // this is a dirty trick, to reverse the order the treemap is
traversed:
    valuables = new TreeMap<Double,
Valuable>(Collections.reverseOrder());
    for (int i = 0; i < piecesOfArt.length; i++) {
        StringTokenizer st = new StringTokenizer(piecesOfArt[i], ",");
        String name = st.nextToken().trim();
        double weight = Double.parseDouble(st.nextToken().trim());
        double price = Double.parseDouble(st.nextToken().trim());
        Valuable valubl = new Valuable(name, weight, price);
        valuables.put(price / weight, valubl);
        //valuables.put(price, valubl);
    }
}
```

In den letzten zwei Zeilen müssen wir uns für unser Gewinnmaximierungskriterium entscheiden, entweder Profit-Density oder Preis. Das ist dann der Schlüssel für unsere TreeMap, nach dem wird also absteigend sortiert. Das ist unsere Greediness.

Dann haben wir noch das Kriterium, dass wir maximal 300kg tragen können, das wir wie folgt umsetzen:

```
    ...
    println("Take the following items:");
    double weight = 0;
    double value = 0;
    for (double key : valuables.keySet()) {
        Valuable val = valuables.get(key);
        if (weight + val.weight <= 300.0) {
            weight += val.weight;
```

```
                value += val.price;
                println("- "+val);
            }
        }

        println("\nTotal weight: " + weight + "kg, total value: €"+value);
```

Wir iterieren also durch unseres TreeMap und fügen alles in unseren Rucksack was noch reinpasst.
Die Resultate sind ganz interessant: wenn wir nach Profit-Density optimieren,

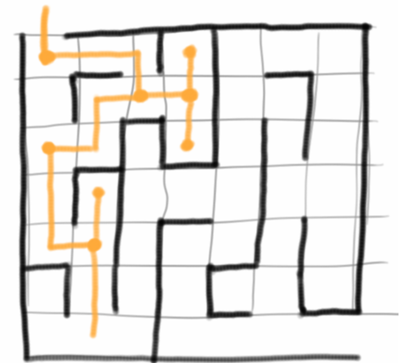
```
Total weight: 105.0kg, total value: €1.815E7
```

dann müssen wir zwar mehr tragen, nehmen aber auch mehr Geld mit nach Hause. Denn wenn wir nach dem Preis optimieren,

```
Total weight: 82.0kg, total value: €1.415E7
```

dann müssen wir zwar etwas weniger tragen, bekommen aber auch nicht so viel Geld dafür.

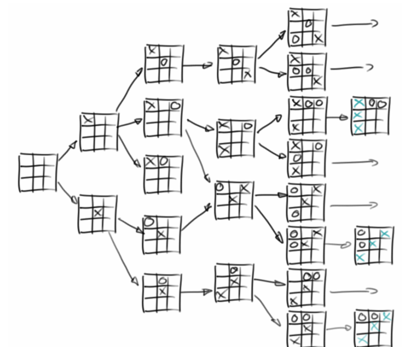
Backtracking



Probleme die sich entweder in Form von Permutationen oder Subsets ausdrücken lassen, können sehr häufig durch den sogenannten *Backtracking* Algorithmus gelöst werden (auf deutsch *Rücksetzverfahren*). Die Idee ist relativ einfach wenn man sie sich am Beispiel eines Labyrinths veranschaulicht: an jeder Kreuzung hat man mehrere Möglichkeiten: man kann nach links, geradeaus oder nach rechts gehen. Beim Backtracking entscheidet man sich dann, z.B. immer erst mal nach links zu gehen. Das macht man so lange bis man entweder am Ziel ist, oder bis es nicht mehr weitergeht. Wenn es nicht mehr weitergeht, dann muss man einen Schritt zurückgehen, denn an der letzten Kreuzung hat man offensichtlich eine falsche Entscheidung getroffen. Man versucht es also mit einer anderen Möglichkeit, z.B. gerade aus. Kommt man da auch nicht weiter, dann kann man es noch rechts versuchen, und geht das auch nicht, dann muss man noch einen Entscheidungpunkt weiter zurückgehen.

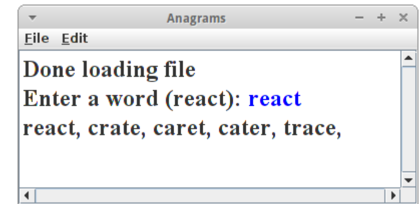
Backtracking geht also alle Möglichkeiten durch, gehört somit zu den Brute-Force Algorithmen und kann deshalb u.U. sehr lange dauern, aber wenn es eine Lösung gibt, dann findet Backtracking sie. Backtracking ist ein rekursiver Algorithmus, und wir können die Vorgehensweise grob so zusammenfassen:

1. unser Problem muss sich irgendwie als eine Reihe von Entscheidungen darstellen lassen;
2. an jedem möglichen Entscheidungpunkt treffen wir eine Entscheidung, am besten immer nach dem gleichen Prinzip (z.B. immer erst mal links gehen), und versuchen so weit zu gehen wie es geht;
3. wenn wir am Ziel angekommen sind, dann sind wir fertig (base case), wenn es aber nicht mehr weitergeht, dann gehen wir zurück zum letzten Entscheidungpunkt, und versuchen es mit einer anderen Entscheidung (recursive case).



Backtracking Algorithmen lassen sich immer als Baum oder als Graph visualisieren, wie z.B. die möglichen Spielzüge beim Spiel TicTacToe. Der Backtracking Algorithmus kommt dann einer Tiefensuche in dem Graphen der möglichen Spielzüge gleich.

Als Brute-Force Algorithmus hat Backtracking zunächst einmal die Tendenz sehr lange zu dauern. Z.B. ein Spiel wie Schach hat viel zu viele Möglichkeiten, als dass man die alle ausprobieren könnte. Aber häufig kann man dem Algorithmus etwas unter die Arme greifen, z.B. durch Pruning oder gewisse Heuristiken, in dem man die am vielversprechenste Wahl trifft. Wir werden bei den Beispielen Anagram, 8-Queens und Sudoku sehen wie man Backtracking implementiert. Auch das Problem des Handlungsreisenden und das Rucksackproblem lassen sich durch Backtracking lösen.



Anagrams

Anagramme sind Wörter mit den gleichen Buchstaben, also z.B. "regal" und "lager", oder "mary" und "army". Die einfachste Möglichkeit Anagramme zu finden, ist einfach alle auszuprobieren, also *Brute-Force*. Dazu bilden wir alle Permutationen mit dem Code aus Kapitel vier, mit einer kleinen Modifikation, wir checken jede gefundene Permutation ob es sich dabei um ein wirkliches Wort handelt:

```
private void permuteBruteForce(String picked, String remaining) {
    // base case
    if (remaining.length() == 1) {
        String tmp = picked + remaining;
        if (lexicon.contains(tmp)) {
            print(tmp + ", ");
        }
    }

    // recursive case
    for (int i = 0; i < remaining.length(); i++) {
        char pick = remaining.charAt(i); // pick a letter
        String front = remaining.substring(0, i);
        String back = remaining.substring(i + 1);
        permuteBruteForce(picked + pick, front + back);
    }
}
```

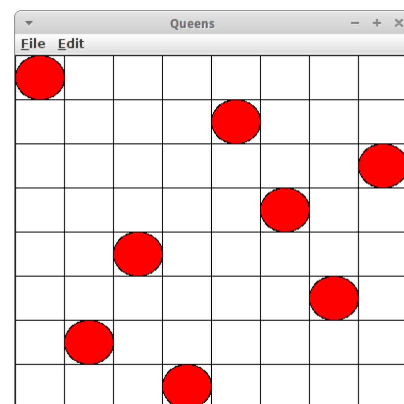
Da wir alle Permutationen durchgehen, hat der Algorithmus ein O(n!) Laufzeitverhalten, also schon für Wörter die mehr als zehn Buchstaben haben, kann das sehr lange dauern.

Im Unterschied dazu versucht der *Backtracking* Algorithmus nur *eine* gültige Lösung zu finden, und sobald er die hat hört er auf:

```
private boolean permuteBacktracking(String picked, String remaining) {
    // base case
    if (remaining.length() == 1) {
        String tmp = picked + remaining;
        if (lexicon.contains(tmp)) {
            print(tmp + ", ");
            return true;
        }
    }

    // recursive case
    for (int i = 0; i < remaining.length(); i++) {
        char pick = remaining.charAt(i); // pick a letter
        String front = remaining.substring(0, i);
        String back = remaining.substring(i + 1);
        if (permuteBacktracking(picked + pick, front + back)) {
            return true;
        }
    }
    return false;
}
```

Die Backtracking Version ist der Brute-Force Variante sehr ähnlich, allerdings eben mit dem Unterschied, dass sie aufhört sobald eine gültige Kombination gefunden wurde.

Das Anagramm Beispiel ist auch sehr gut geeignet um zu erklären was *Pruning* ist: das Wort kommt aus der Baumschule, und hat mit dem Zurückschneiden von Bäumen zu tun. Da wir ja beim Backtracking einen Baum (oder Graphen) von Entscheidungen durchlaufen, stellt sich die Frage ob wir alle Äste wirklich bis zum Ende durchgehen müssen, oder ob wir manche Äste abschneiden (prune) dürfen. Nehmen wir z.B. Anagramme zum Wort "mary": Wir wissen, dass es in der englischen Sprache keine Worte gibt die mit "mr" beginnen. D.h. wir brauchen keine Permutation von Worten die mit "mr" beginnen weiter zu ermitteln und wir können den Teil des Entscheidungsbaumes "abschneiden". In der Praxis könnte man so etwas mit der *Trie* Datenstruktur umsetzen, die man mit allen Wörtern der englischen Sprache befüllt.



8-Queens

Das *Eight Queens Puzzle*, auch Damenproblem genannt, ist ein weiteres Beispiel, das sich mit dem Backtracking Algorithmus lösen lässt. In dem Damenproblem geht es darum acht Damen auf einem Schachbrett so zu platzieren, dass sie sich nicht gegenseitig bedrohen. Die Dame im Schach kann sowohl auf den Horizontalen als auch auf den Vertikalen und den Diagonalen schlagen.

Man sieht eigentlich sofort, dass in einer Reihe jeweils nur eine Dame stehen kann. Überlegen wir uns weiter wie wir hier den Backtracking Algorithmus einsetzen können: Wir beginnen mit der ersten Dame in der ersten Reihe. Im Prinzip haben wir acht Möglichkeiten, aber es macht wohl Sinn links anzufangen, und sie in die erste Spalte zu setzen. Kommen wir zur zweiten Dame: die muss in der zweiten Reihe sein. Auch hier gibt es wieder acht Spalten auf die wir die zweite Dame setzen könnten. Wir beginnen links, und versuchen einfach eine Spalte nach der anderen, bis wir eine finden in der die zweite Dame nicht von der ersten bedroht wird. Das ist die dritte Spalte. Wir machen so weiter mit der dritten Dame, etc., bis wir entweder alle Damen gesetzt haben, oder bis wir eine Dame in keine Spalte setzen können, weil sie überall bedroht wird. Dann müssen wir zurück gehen und eine der vorherigen Damen woanders hinsetzen. Das ist das Backtracking.

Um das in Code umzusetzen, benötigen wir erst einmal die Positionen der Damen:

```
private Point[] queens = new Point[NR_OF_QUEENS];
```

Wir beginnen mit der ersten Dame, Dame Nummer Null, und platzieren sie:

```
placeQueen(0);
```

Die Methode *placeQueen()* versucht nun rekursiv eine Dame nach der anderen zu setzen:

```
private boolean placeQueen(int n) {
    // stop condition, we placed all queens:
    if (n >= NR_OF_QUEENS) {
        return true;
    }

    boolean[][] tmp = markThreatenedFields();
    // every possible choice among the columns in this row
    for (int i = 0; i < NR_OF_QUEENS; i++) {
        // check if point is safe
        if (isSafe(i, n, tmp)) {
            queens[n] = new Point(i, n);

            if (placeQueen(n + 1)) {
                return true;
            } else {
                queens[n] = null;
```

```
                }
            }
        }
        return false;

    }
```

Das Abbruchkriterium ist ganz einfach, wenn alle Damen gestetzt sind, sind wir fertig. Andernfalls, markieren wir erst einmal alle Felder die momentan von Damen bedroht werden, das macht die Methode *markThreatenedFields()*. Dann versuchen wir die momentane Dame auf eine der acht möglichen Spalten zu setzen. Die Methode *isSafe()* sagt uns ob die Spalte sicher ist. Falls ja, dann platzieren wir die Dame dort, und versuchen die nächste Dame zu platzieren. Wenn das gelingt super, dann war die Wahl der Spalte o.k., falls das aber nicht gelingt, dann war die Spalte wohl nicht gut, deswegen machen wir die Wahl mit *queens[n] = null;* rückgängig.

Die Methode *isSafe()* ist ganz einfach,

```
    private boolean isSafe(int x, int y, boolean[][] tmp) {
        return !tmp[x][y];

    }
```

und die Methode *markThreatenedFields()* geht einfach eine Dame nach der anderen durch,

```
    private boolean[][] markThreatenedFields() {
        boolean[][] tmp = new boolean[NR_OF_QUEENS][NR_OF_QUEENS];
        for (int i = 0; i < NR_OF_QUEENS; i++) {
            if (queens[i] != null) {
                markHorizontal(queens[i], tmp);
                markVertical(queens[i], tmp);
                markDiagonal1(queens[i], tmp);
                markDiagonal2(queens[i], tmp);
            }
        }
        return tmp;

    }
```

und markiert alle Felder die bedroht werden mit *true*. Z.B. *markHorizontal()* macht das folgendermaßen:

```
    private void markHorizontal(Point queen, boolean[][] tmp) {
        for (int x = 0; x < NR_OF_QUEENS; x++) {
            tmp[x][queen.y] = true;
        }

    }
```

So funktioniert ein klassischer Backtracking Algorithmus.

Sudoku

Bei Sudoku handelt es sich um ein Zahlenrätsel. In der 9x9 Version geht es darum die Zahlen 1 bis 9 so zu verteilen, dass in einer Reihe, einer Spalte und in einem Block jede Zahl nur einmal vorkommt. Es gibt auch eine 12x12 Version, eine 4x4 Version und die 1x1 Spezialversion für Männer [6]. Sudoku ist auch wieder eine sehr schöne Anwendung für den Backtracking Algorithmus [7].

Die Zahlen in unserem Sudoku stellen wir als 9x9 Array von Ganzzahlen dar:

```
    private int[][] grid;
```

wir initialisieren es in dem wir das zu lösende Sudoku Rätsel aus einer Datei lesen:

```
grid = readGridFromFile(SUDOKU_FILE_NAME);
```

unbesetzte Felder werden dabei auf Null gesetzt. Nach der kurzen Vorarbeit können wir uns dem Lösungsalgorithmus zuwenden:

```
private boolean solveSudoku() {
    Point p = new Point(0,0);

    if (!findUnassignedLocation(grid, p)) {
        return true;
    }

    for (int num = 1; num <= SIZE_OF_SUDOKU; num++) {
        if (noConflicts(grid, num, p.y, p.x)) {
            grid[p.y][p.x] = num;
            if (solveSudoku()) {
                return true;
            }
            grid[p.y][p.x] = 0;
        }
    }
    return false;

}
```

Unser Abbruchkriterium ist durch die *findUnassignedLocation()* Methode gegeben: Falls es also im ganzen Grid keine '0' mehr gibt, dann sind wir fertig. Andernfalls, gibt die Methode allerdings den Punkt zurück um den wir uns als nächstes kümmern sollten, denn der Punkt *p* wird als Referenz übergeben, und in der Methode verändert:

```
private boolean findUnassignedLocation(int[][] grid, Point p) {
    for (int i = 0; (i < grid.length); i++) {
        for (int j = 0; (j < grid.length); j++) {
            if (grid[i][j] == 0) {
                p.y = i;
                p.x = j;
                return true;
            }
        }
    }
    return false;

}
```

Im nächste Schritt versuchen wir dann an diesem Punkt *p* jede der möglichen Zahlen, also 1 bis 9, zu setzen. Ob das geht sagt uns die Methode *noConflicts()*:

```
private boolean noConflicts(int[][] grid, int num, int row, int col) {
    // first check rows
    for (int i = 0; i < grid.length; i++) {
        if (grid[row][i] == num) {
            return false;
        }
    }
    // next check cols
    for (int i = 0; i < grid.length; i++) {
        if (grid[i][col] == num) {
            return false;
        }
```

```
        }
        // finally check blocks
        // first find relevant block:
        int blockLength = (int) Math.sqrt(grid.length);
        int j0 = col / blockLength;
        int i0 = row / blockLength;
        for (int i = 0; i < blockLength; i++) {
            for (int j = 0; j < blockLength; j++) {
                if (grid[i0 * blockLength + i][j0 * blockLength + j] == num)
    {
                    return false;
                }
            }
        }
        return true;
    }
```

Die prüft einfach die Sudoku Regeln, dass in jeder Reihe, jeder Spalte und in jedem Block jede Zahl nur einmal vorkommen darf. Falls es also gelingt, dann versuchen wir es weiter mit der nächsten freien Stelle. Ansonsten gehen wir zurück (backtracking) und machen unsere letzte Wahl rückgängig.

Randomized Algorithms

Eine besondere Klasse sind Algorithmen die den Zufall zu Hilfe nehmen. Erst einmal scheint sich das zu widersprechen, ein Algorithmus ist doch eine genaue Vorschrift wie ich ein Problem löse, und das Gegenteil ist einfach zufällig mal rumprobieren bis man eine Lösung findet. Wie so häufig ist es der goldenen Mittelweg der zum Erfolg führt. Es gibt auch eine ganze Menge von Problemen, die sich überhaupt nur mit Algorithmen lösen lassen die den Zufall verwenden, dafür gibt es zwei Klassen von Algorithmen, die *Monte Carlo* und die *Las Vegas* Algorithmen.

Review

Im ersten Kapitel haben wir bereits Bekanntschaft mit dem Zufall gemacht: das erste Mal als wir Pi ausgerechnet haben. Das ist ein Algorithmus der sich Zufall zuhilfe nimmt um etwas zu berechnen. Auch wenn wir Leute in einem Stadium zählen mittels der Methode, dass alle Leute deren Nachname mit 'A' beginnt aufstehen sollen, dann machen wir ein *Sampling*, und gehen aber davon aus, dass die Nachnamen alle gleichhäufig im Alphabet vorkommen. Das ist auch eine Annahme die mit Zufallsverteilung der Nachnamen zu tun hat. Im ersten Kapitel haben wir auch gesehen wie wir Pseudo-Zufallszahlen erzeugen können, mit Hilfe von Lehmer's Algorithmus. Wir haben auch gesehen, wie man schlechte Zufallszahlen identifizieren kann.

Weitere Algorithmen die wir bereits gesehen haben und die sich in der einen oder anderen Form des Zufalls bedienen, waren:

- Pi
- RandomArt
- Tree
- Maze
- Lightning
- PlasmaCloud
- Shuffle
- QuickSort

Man könnte meinen Algorithmen die mit Zufall arbeiten liefern immer nur Näherungen, dass stimmt aber nicht, denn QuickSort verwendet Zufall liefert aber eine genaue Lösung.

LoadBalancing

Im zweiten Kapitel haben wir uns mit dem Thema *LoadBalance* kurz beschäftigt. Damals haben wir neue Tasks auf denjenigen Server verteilt, der die geringste Last hat. Nicht immer weiss man aber welche Last welcher Server hat. Dann gibt es prinzipiell zwei Möglichkeiten: Round Robin [8], also einer nach dem anderen,

```
// round robin assignment of task
private void addNewServerTask() {
    int serverNr = currentTaskNr % NR_OF_SERVERS;
    currentTaskNr++;
    serverQueues[serverNr].add("Task Nr." + currentTaskNr);

}
```

oder zufällige Verteilung,

```
// random assignment of task
private void addNewServerTaskRandom() {
    int serverNr = (int) (Math.random() * NR_OF_SERVERS);
    currentTaskNr++;
    serverQueues[serverNr].add("Task Nr." + currentTaskNr);

}
```

Letzterer ist ein Algorithmus der sich den Zufall zu Hilfe nimmt.

Minimum, Maximum and Average

Im Kapitel "Algorithmic Analysis" haben wir gesehen, dass die Berechnung von Minimum, Maximum und Average linear in der Zeit ist, also O(n), es sei denn die Daten sind sortiert. Wenn wir jetzt aber nicht sortieren wollen, und uns eine ungefähre Antwort, also eine Näherung, genügt, dann geht das auch schneller.

Nehmen wir an wir wollen den ungefähren Durchschnitt für eine Array von Ganzzahlen berechnen:

```
int[] arr = { 5, 55, 2, 7, 45, 3, 1, 8, 23, 12 };
```

Eine Möglichkeit ist einfach das erste, das mittlere und das letzte Element zu nehmen und den Durchschnitt zu bilden:

```
double average = ( arr[0] + arr[arr.length/2] + arr[arr.length-1] ) /
3.0;
```

Ein andere Möglichkeit ist drei zufällige Elemente (drei Samples) zu wählen und den Durchschnitt zu bilden:

```
private static double averageRandom(int[] arr) {
    int total = 0;
    int len = arr.length;
    for (int i = 0; i < 3; i++) {
        total += arr[(int) (len * Math.random())];
    }
    double average = total / 3.0;
    return average;

}
```

Das funktioniert für beliebig große Arrays, und je mehr Samples wir nehmen, desto genauer wird es. Die Geschwindigkeit dieses Algorithmus hängt nur von der Anzahl der Samples ab, und nicht von der Größe des Arrays, d.h. er wird in der Regel O(1) Laufzeitverhalten haben.

Monte Carlo and Las Vegas

Was Zufalls-Algorithmen angeht unterscheidet man grundsätzlich zwei Klassen: die *Monte Carlo* und die *Las Vegas* Algorithmen [9,10]. Der Hauptunterschied liegt darin, dass man bei den *Las Vegas* Algorithmen verifizieren kann, dass die Antwort richtig ist. QuickSort ist z.B. solch ein Algorithmus. Bei *Monte Carlo* Algorithmen kann man das nicht, z.B. die Berechnung von Pi fällt in diese Kategorie.

Interessant ist auch, dass es eine ganze Reihe von Problemen gibt, die sich sinnvoll überhaupt nur mit Zufalls-Algorithmen lösen lassen. Das ist sehr häufig bei Problemen der Fall die ein exponentielles Laufzeit Verhalten haben. Die kann man zwar im Prinzip lösen, allerdings nur wenn man sehr, sehr, sehr viel Zeit hat. In der Regel hat man aber nicht so viel Zeit, und deswegen ist man froh, wenn man zwar keine exakte, aber doch eine annehmbare Lösung hat. Z.B. das Traveling Salesman Problem [11], das Scheduling Problem und das Planning Problem fallen alle in diese Kategorie.

Algorithmen die versuchen eine annehmbare Lösung zu finden, meist mit Hilfe des Zufalls arbeiten, sind z.B.

- Genetic Programming [12]
- Particle Swarm Optimization [13]

Vielleicht sehen wir den einen oder anderen etwas später noch.

Review

In diesem Kapitel haben wir etwas von Brute Force, Greedy, Back Tracking und Randomized Algorithmen gehört. Zu den Anwendungen gehört das Geld Zählen, das Planen, das Stehlen und das Spielen.

Fragen

1. Was ist ein “gieriger” (greedy) Algorithmus? Erklären Sie, wie diese funktionieren, evtl mit einem Beispiel.
2. Was ist ein Rucksackproblem? Beschreiben Sie es mit einem Beispiel.
3. Geben Sie zwei Beispiele für einen Greedy Algorithmus.
4. Welchen Algorithmus würden Sie verwenden, um das 4 Queens Problem zu lösen?
5. Liste Sie alle möglichen Permutationen der drei Buchstaben "car".
6. Erläutern Sie den Unterschied zwischen Combinations, Permutations und Subsets. Geben Sie evtl je ein Beispiel an.
7. Können Sie das Konzept des "Pruning" erklären? Warum ist das überhaupt so wichtig?
8. Welchen Algorithmus würden Sie benutzen, um einen Weg aus einem Labyrinth zu finden? Bitte erklären Sie grob, wie Ihr Algorithmus funktionieren würde.

Referenzen

Fast fertig, zum Schluss noch ein paar Referenzen.

[1] Huffman coding, https://en.wikipedia.org/wiki/Huffman_coding

[2] Greedy coloring, https://en.wikipedia.org/wiki/Greedy_coloring

[3] Algorithmen und Datenstrukturen, Gunter Saake und Kai-Uwe Sattler

[4] Jon Kleinberg, Eva Tardos, Kevin Wayne, Algorithm Design, GREEDY ALGORITHMS I, https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/04GreedyAlgorithmsI.pdf

[5] 10 Must-See Works Of Art At The Louvre, Huffington http://www.huffingtonpost.com/2012/08/10/happy-birthday-museum-cen_n_1761094.html

[6] Private communication, Weber-Wulff, D.

[7] Computer Science II: Programming Abstractions, Julie Zelenski, Stanford, https://see.stanford.edu/materials/icspacs106b/Lecture11.pdf

[8] Round-robin scheduling, https://en.wikipedia.org/wiki/Round-robin_scheduling

[9] Monte Carlo algorithm, https://en.wikipedia.org/wiki/Monte_Carlo_algorithm

[10] Las Vegas algorithm, https://en.wikipedia.org/wiki/Las_Vegas_algorithm

[11] Travelling salesman problem, https://en.wikipedia.org/wiki/Travelling_salesman_problem

[12] Genetic programming, https://en.wikipedia.org/wiki/Genetic_programming

[13] Particle swarm optimization, https://en.wikipedia.org/wiki/Particle_swarm_optimization

[14] Brute-Force and Greedy Algorithms, http://www.brpreiss.com/books/opus5/html/page433.html#31938

Epilogue

Das war jetzt echt viel Arbeit.

Index

A
Adjacent141f., 164
Adventure51, 151f.
Anagramm194f.
Ancestor108, 136
ArithmeticExpression23, 32, 36ff., 68, 129
ArrayList19ff., 23, 25f., 28ff., 38ff., 45f., 50f., 53ff., 62, 131, 133f., 151, 164, 174

B
Binärbaum121, 125, 130, 134
BinaryNode111, 116, 120ff., 129ff.
BinarySearch105, 112
BinaryTree107, 111, 116f., 120ff., 125, 131f., 134
Binomialkoeffizienten71f.
Breitensuche143f., 146, 150

C
Combination72, 80, 92, 200
Composition57, 59f.

D
Datencontainer19f., 61
Decision112, 116f., 125, 134, 137
Descendant108, 136
Dictionary44, 49, 52, 56, 131, 133, 173
DiGraph149, 151f., 156
Dijkstra146f., 150, 154, 157, 163, 165f., 186
Dynamisch21, 81ff., 89ff., 185

E
Entscheidung16, 71, 108, 112, 115ff., 134, 191, 193, 195
Entscheidungsbaum112, 117, 134, 195

F
Fibonacci72, 89ff., 96, 110, 120f.
FileTree118f.
FlightFinder154, 160, 163

G
Generalization57ff.
GraphEdgeList142, 149, 151ff., 158, 161
Greed1, 146, 186ff., 191f., 200f.

H
Handlungsreisenden186, 194
HashCode45f.
HashList55
HashMap14, 44ff., 49ff., 55ff., 62, 151, 158f., 161f.
HashSet47, 52, 54, 105, 114, 133
HeapSort106
Histogram24, 57

I
Insertion100, 103, 106, 117
IODialog9, 179
Iterator48

K
Kaffeeautomaten186, 188
Knapsack191
Kombination70, 72f., 181, 186f., 191, 195
Kreditkarten12f., 61
Kreuzworträtsel176f.

L
Labyrinth74, 193, 200
LevelOrder109, 111, 116, 125, 127f.
Levenshtein170, 183
Lightning76, 80, 198
LinkedList19ff., 27ff., 34f., 39f., 53, 62, 133, 164
List1, 14, 16f., 19ff., 34f., 38ff., 44ff., 58ff., 70, 72, 95, 105, 113ff., 124, 131, 133f., 140, 142, 144, 149ff., 158f., 161ff., 174f., 182, 188, 191, 200
LoadBalance35, 199

M
Mandelbrot14, 17
Map1, 14, 19, 43ff., 55ff., 106, 114, 140, 146, 151, 154, 156ff., 166, 176, 187f., 192f.
Matching40, 169f., 183
MergeSort100ff.
Mindesten55, 147, 156f., 181
Mondrian73f., 79f.

N
Node27ff., 108ff., 115f., 118ff., 136, 150, 153, 158f., 172ff.

O
OrderedTree110, 115, 118f., 122ff., 129

P
Pascal71ff., 80, 139
Permutation70, 80, 92, 95, 193ff., 200
Phonetische167, 171, 173
PostOrder109f., 118f., 125
PreOrder109f., 122, 125
Prim11, 147, 150, 155f., 166, 172, 186
Priority22f., 146ff., 189
PriorityQueue22f., 147f., 189
Projektmanagement2, 149, 156
Prüfziffer12f.

Q
Queue19, 22f., 34ff., 39ff., 147f., 189, 199
QuickSort102f., 106, 114, 117, 198, 200

R
RandomArt36, 198
RandomGenerator9f., 15f.
Recursion1, 63, 71, 79f., 83f., 86f., 90, 94
Rekusion63, 69, 83f.
Ringpuffer39
Rucksack186, 191, 193f., 200
Rucksackproblem186, 191, 194, 200

S
Scheduling149, 188, 191, 200f.
Schlange22f., 34
SelectionSort95, 117
Sierpinski73, 77, 79f.
Sortieralgorithmen97, 99f., 103, 114, 117
Sortieren52, 97f., 100, 103ff., 114f., 199
Sortierverfahren106, 114
Spellchecker52, 61, 167, 170
Stack19, 21ff., 30ff., 37f., 40f., 129f., 175
Stopword53, 182
SubGraph141, 147, 158
Subsequence171, 178, 183
Suchmaschine172, 175, 182
Synonym44, 151f., 158

T
Teilbaum108, 112f.
Telefonbuch44f.
Tiefensuche143f., 150, 193
Topological150, 157, 166
Topologisch149f., 157
TreeSet47, 52, 113f., 133

U
Unvollständigkeitssatz78

V
Vertex140ff., 146f., 149, 151ff., 161, 163f.

W
Warteschlange22f.
Wildbienen118, 134, 137
Wörterbuch44, 55, 59ff., 131, 173f., 176

Z
Zufallszahlen10ff., 17, 52, 98, 198